NASA Contractor Report 187522

(NASA-CR-187522) STRUCTURED REPRESENTATION FOR REQUIREMENTS AND SPECIFICATIONS (Boeing Military Airplane Development) 93 p CSCL 09B

N91-2[illegible]

Unclas

G3/61 0026000

Structured Representation for Requirements and Specifications

Gene Fisher
Deborah Frincke
Dave Wolber
University of California,
Davis, California

G. C. Cohen
Boeing Military Airplanes
Seattle, Washington

NASA Contract NAS1-18586
July 1991

NASA

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665-5225

Preface

This document was generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems suitable for Fly-By-Wire Applications, Task Assignment 2. Task 2 is associated with a formal representation of requirements and specifications. In particular, this document contains results associated with the development of a Wide-Spectrum Requirements Specification Language (WSRSL) that can be used to express system requirements and specifications in both stylized and formal forms. Included with this development are prototype tools to support the specification language. In addition a preliminary requirements specification methodology based on the WSRSL has been developed. Lastly, the methodology has been applied to an Advanced Subsonic Civil Transport Flight Control System.

The NASA technical monitor for this work is Sally Johnson of the NASA Langley Research Center, Hampton, Virginia.

The work was accomplished at Boeing Military Airplanes, Seattle, Washington, and the University of California, Davis, California. Personnel responsible for the work include:

Boeing Military Airplanes:

D. Gangsaas, Responsible Manager

T. M. Richardson, Program Manager

G. C. Cohen, Principal Investigator

University of California:

Dr. Gene Fisher, Chief Researcher

Deborah Frincke, PhD Candidate

Dave Wolber, PhD Candidate

Contents

List of Figures

List of Tables

1 Introduction

This report describes our results for the project on the Structured Representation for Requirements and Specifications. Our overall impression of the project is that it has been successful in accomplishing the goals that were set out in the original proposal. To summarize our results, we have:

- developed a Wide-Spectrum Requirements Specification Language (WSRSL) that can be used to express system requirements and specifications in both stylized and formal forms
- developed prototype tools to support the specification language
- developed a preliminary requirements specification methodology based on our wide-spectrum language
- applied the methodology to translate one third of the document "An Example of Requirements for Advanced Subsonic Civil (ASCT) Flight Control System Using Structured Techniques" [CM 91]

Our major contributions have been improvements in the organization and formality of the ASCT document. The WSRSL provides the means to organize a requirements specification in a modular fashion, and eliminate much of the informal prose that is found in documents such as ASCT. Since WSRSL is a formal language, it can be formally analyzed for syntactic correctness, completeness, and consistency. In addition, some aspects of semantic correctness can be analyzed and verified.

Another significant contribution is the development of tools to permit a WSRSL document to be viewed electronically. Electronic viewing is particularly useful in large documents, where the linear medium of the printed page often hinders the reader from locating concepts of interest and understanding the connectivity of the different document components. Our tools provide a means to browse the document in hypertext style, and to view graphical dataflow diagrams that depict the connectivity of the operations defined in a WSRSL document.

The remainder of the report describes our results in further detail. Section 2 presents an overview of the Wide-Spectrum Requirements Specification Language (WSRSL). Section 3 describes the more formal aspects of WSRSL. Section 4 describes the requirements specification methodology based on WSRSL. Section 5 describes the two tool prototypes — the browser and dataflow editor.

2 Overview of the Wide-Spectrum Requirements Specification Language

Our Wide-Spectrum Requirements Specification Language (WSRSL) is a hybrid of the features found in several popular specification languages and systems. The basic features for specifying the hierarchical structure of objects and operations are similar to features in such languages as Revised Special [C* 86], SADT (Structured Analysis and Design Technique) [RS 77]], PSL (Problem Statement Language) [TH 77], and RSL (Requirements Specification Language) [GMB 82].

The purpose of the language is to describe precisely the external structure of the objects and operations for specifying aircraft systems. While languages such as SADT and PSL have some good underlying concepts for how to structure a specification, they are notorious for a lack of formality, making it difficult to be completely precise in a specification. Accordingly, the WSRSL includes features for formal specification, as will be described in subsections below.

Using our WSRSL, a specification consists of two parts: (1) an ***object-oriented*** part, and (2) a ***function-oriented*** part. These two parts form two different ***views*** of the same system. One view presents the system from the perspective of the objects (i.e., data), the other from the perspective of the operations (i.e., functions). Neither view is ***the*** correct one – they both convey the same meaning in different ways. Depending on the natural orientation of the system being specified, one form of view may be the more natural form of specification. Specifically, for the so-called "transaction-oriented" systems [YC 79], the object-oriented view is typically more natural. Conversely, for the "transform-oriented" systems [YC 79], the operation-oriented view is typically the more natural.

Whichever is the more natural view, a specification should always contain both. In this way, the two views provide a form of cross checking on specification consistency and completeness. Something that may have been overlooked in the function-oriented view may show up naturally in the object-oriented view, and vice versa.

2.1 Underlying Pervasive Principles

The WSRSL described here, as well as similar requirements languages such SADT and RSL, share some common underlying principles. These principles are:

1. ***Entity/Relation Model*** – the specification universe is viewed as comprised of ***entities*** which have well-defined ***relations*** to other entities in the specification universe. In particular, there are two primary types of entities to consider – ***objects*** and ***operations***.
2. ***Attribute/Value Pairs*** – in addition to relations to other entities, an entity may have zero or more general ***attributes*** that describe further the entity's properties.
3. ***Hierarchy*** – the primary relation between entities of the same type is ***hierarchy***. That is, an entity is composed hierarchically of subentities, which may in turn be further hierarchically decomposed.
4. ***Four Composition Primitives*** – When an entity is decomposed into subentities, four specific composition forms are used. These are:
   (a) *AND* composition: an entity is composed as a ***heterogeneous collection*** of subentities
   (b) *OR* composition: an entity is composed as a ***selected one of*** a heterogeneous collection of subentities

   (c) ***repetitive*** composition: an entity is composed as a ***homogeneous collection*** of subentities
   (d) ***Recursive*** composition: entity/subentity hierarchical decomposition may be ***recursive;*** i.e., a subentity may be the same as its (grand ...)parent entity(ies).

5. ***Basic Composition Modifiers*** – further constraints can be placed on composed collections of subentities. Typical of such are the following:

   (a) ***Ordered*** – formally, a ***partial ordering*** relationship must be defined between subentities
   (b) ***Duplicate Subentities*** – formally, an ***equivalence relationship*** must be defined between subentities
   (c) ***Indexable*** – an ***index key*** must be specified for subentities

6. ***Class/Subclass/Instance Composition*** – a secondary form of hierarchical relation is that of ***class/instance***. An entity ***class*** defines a generic entity form, i.e., a template. An entity ***instance*** can then be defined as a more ***specialized***[1] member of a class; the instance ***inherits*** the attributes of the parent class of which it is a specialization, plus defines additional instance-specific attributes. The class/instance hierarchy can have an arbitrary number of levels; i.e., an instance of one class may be a class itself, in which class it is called a ***subclass***.

7. ***Object/Operation Duality*** – the primitive composition and relational forms apply equally to both objects and operations. I.e., both objects and operations can be decomposed hierarchically, with general attribute/value pairs, and in class/subclass hierarchies. Object/operation duality manifests itself in interesting ways, as will be discussed in future reports.

2.2 Specifying Objects

An object is specified in a fixed format showing its components and other attributes. The general form is as follows[2]:

object ***name*** **is**
    components: ***list of subobjects***
    operations: ***list of operations, including their argument types***
    [**description:** {***free-form text***}]
    [**example:** {***free-form text/graphics***}]
    [`other attributes:` ***Other useful information as appropriate***]
end ***name***

For example:

object `Mission` **is**
    components: `TaxiInOut` **and** `TakeOff` **and** `Climb` **and**
        `Cruise` **and** `Descent` **and** `Approach` **and** `Landing`

[1]a synonym for the *class/subclass* relation is *generalization/specialization*

[2]boldface terms are keywords, italic terms are variables, optional terms are enclosed in square brackets [...]

```
    operations:
        Navigate: Mission -> TargetFlightPlan
        ... ( additional operations) ...
    description: {A Mission is the main object of the ASCT Flight Control
        System.  Its components represent each of the main phases of a
        controlled flight.  ... .}
end Mission

object TakeOff is
    components: AccelerationPhase and RunwayDeparture
    operations:
        AccelerateToTakeOff:  AirCraftState, SpeedControls -> AirCraftState
        DepartRunway: AirCraftState, LiftOffControls -> AirCraftState
    description: {The TakeOff phase of a Mission covers the period from
        the completion of taxiing to the beginning of the climbing phase.}
end TakeOff
```

2.3 Specifying Operations

An operation specification is much like an object specification, in a fixed format of components and other attributes. Here is the general form:

```
operation name is
    components:  list of suboperations
    inputs:  list of objects
    outputs:  list of objects
     [ description:  {free-form text/graphics} ]
     [ other attributes:  Other useful information as appropriate ]
end name
```

For example:

```
operation PerformTakeOff is
    components: Accelerate and Depart
    inputs: AircraftState, SpeedControls
    outputs: AircraftState
    description: {The aircraft has received clear-to-depart signal from
        the tower and proceeds to accelerate down the runway and depart.}
    precondition: ClearToDepart = true and . . .
    postcondition: AirCraftState = Climbing  and . . .
    location: Runway
    agent: Pilot
    instrument: SpeedControlDevice
end PerformTakeOff
```

2.4 Specifying Modules

The ***module*** is the WSRSL packaging construct. WSRSL modules serve precisely the same purpose as modules in other command specification and programing languages, such as EHDM [C* 86], Modula-2, Ada, and many others. The general form for a module is the following:

```
[ module-type ] module name is
    [ attribute declarations ... ]
    [ import-export declarations ... ]
    [ state-variable definitions ... ]
    [ entity definitions ... ]
    [ formal definitions ... ]
end name
```

The addition of modules makes it possible to include ***state-variable definitions*** within a WSRSL specification. State variables can be declared as any object type defined in the module. These variables represent specific object instances that can be reasoned about using the theory of states defined in Revised Special, as discussed in a later section of the report.

The ***entity definitions*** in a module are a collection of functionally-related object and operation definitions. The "functional relatedness" is not enforced by the language. Rather, the specifier determines which entity definitions are sufficiently logically related so as to appear in the same module. Modularization techniques have been discussed extensively in the literature on specification and programming languages.

Modules also include ***formal definitions***. These provide the fully formal specification of:

- operation preconditions and postconditions
- object equations
- module invariants

As described in the next subsection, each of these specification components is mapped to an underlying Revised Special construct.

WSRSL documents can optionally be organized into several types of modules ***definition***, ***formal***, ***link***, ***dataflow***, ***implementation***, and ***display***. These different types of modules serve the same general purpose as the two-part modules in such languages as Modula-2 and Ada. Namely, a single module can be subdivided into separate parts to further clarify modular structure. In the two-part module schemes, the division is typically between definition and implementation parts. The definition part holds the abstract specification and the implementation part defines the concrete representations. This concept of multi-part module is extended in WSRSL to allow for several additional parts, as outlined in Table 1.

The subdivision of a module into two or more of these parts is optional. That is, all of the the information in the separate parts may be contained in a single module.

2.5 Specifying Attribute Fields

In the general formats shown above, objects and operations may contain user-defined fields that specify other named attributes. Examples of user-defined attributes are the "agent", "location", and "instrument" fields shown in the definition of operation PerformTakeOff.

Module Part	Contents
definition	object and operation definitions
formal	optional separation of axioms and pre/post conds
link	textual specification of browser hyperlinks (see Section 5 below)
dataflow	textual specification of browser hyperlinks (see Section 5 below)
implementation	a concrete implementation description (e.g., a software program or hardware description)
display	textual specification of graphical representations of entities (see proposal for follow-on project)

Table 1: WSRSL Module Types

The contents of these fields can be either free-form text or the names of other entities. The free-form text is specified syntactically as a comment — any text enclosed in braces "{" and "}". User-defined attributes provide the means to systematically specify additional properties of an object or operation. In the PerformTakeOff operation, for example, rather than having a single lengthy **description** field that mentions agent, location, and instrument, these three attributes are given names, to provide a more systematic style of description. Naming attribute fields is particularly useful to provide descriptive consistency. Defining a specific set of attributes that are shared by a class of entities more clearly describes the common characteristics of the class.

When the value of a user-defined attribute is an entity name, rather than a comment, a formal relation is defined between entities. For example, in the Takeoff operation, the attribute specification "agent: Pilot" establishes a formal relation ***agent-of*** between operation PerformTakeOff and object Pilot. These relations can be used in the formal part of a requirements specification to identify classes of objects by their relationships.

To allow user-defined attributes to be checked for consistency, each attribute must be explicitly declared at the top of the module in which it is used. For example, in order to use the agent, instrument, and location attributes in the definitions of PerfomTakeOff and other operations, they must be declared as follows at the top of the module in which the PerformTakeOff operation appears:

```
module FlyMission
  operation field agent
  operation field instrument
  operation field location
    ...
end FlyMission
```

2.6 Advanced Features

2.6.1 Classes

An object or operation definition may be specified as a ***class***. A class definition is a general template for an object or operation, of which more specific instances can be declared. For example,

```
object class Mission is
    components: TaxiInOut and TakeOff and Climb and Cruise
           and Descent and Approach and Landing
    description: {A generic Mission without consideration of special
       circumstances, such as inclement weather or special cargo handling.}
end Mission

object ColdWeatherMission instance of Mission is
    components: PreFlightDeicing and MidCourseFlightAdjustments
end ColdWeatherMission

object SpecialCargoMission instance of Mission is
    components: PreFlightCargoCheck and MidFlightCargoCheck
              and PreLandingCargoCheck
end SpecialCargoMission
```

This example specifies that the general class of all Missions has the same components from TaxiInOut to Landing. Special instances of Missions are differentiated by additional Mission phases, such as specialized pre/post flight checks. Each of the specialized instances of a Mission ***inherits*** the generic Mission's components, plus adds specialized components such as PreFlight-Deicing, MidCourseFlightAdjustments, etc. Similar forms of specialization could be specified for further different forms of Mission. Note that in an instance definition, the inherited components are not repeated – only the additional components are specified.

Operations are inherited along with components. Specialized operations are defined via ***overloading***, as discussed below. For example,

```
object class obj1 is
    components: a,b,c
    ops: op1(a,b,c):obj1
end obj1

object obj2 instance of obj1 is
    components: d
    ops: op1(a,b,c): obj2
end obj2

operation op1 is
```

```
    inputs: a,b,c
    outputs: obj1
end op1

operation op1 is
    inputs: a,b,c
    outputs: obj2
    remark: signature of this specialized version of op1 distinguishes
              it from the op1 with coarity of obj1
end op1
```

2.6.2 Name Qualification and Overloading

In large specifications, there arises the common problem of ***name space conflicts.*** That is, it may be natural or convenient to give the same name to two or more entities, when those entities are part of another. For example, consider the following two object definitions

```
object Aircraft is
  components: ..., State, ...
  operations: ...
   ChangeState(State) -> State
    ...
end Aircraft

object Mission is
  components: ..., State, ...
    ...
  operations: ...
   ChangeState(State) -> State
end Mission
```

Here the two objects both have a component named "State", which is a natural name in both cases. The potential problem is that the State object that is the component of an Aircraft is probably quite different than the definition of the State object that is the component of a Mission. In order to allow two different definitions of State, entity names may be explicitly qualified with the name(s) of parent entity(ies). For example, in this case the two different definitions for State would be defined as follows:

```
object Aircraft.State is
  components: ...
  operations: ...
    ...
end Aircraft.State

object Mission.State is
  components: ...
```

```
   operations: ...
      ...
 end Mission.State
```

Here the names of the State objects being defined are disambiguated with the names of the respective parent entities appearing as qualifying prefixes. In general, a qualifying prefix is of the form

entity-name. ... .entity-name.base-name

where each ***entity-name*** to the right of a "." must be a component of the entity immediately to the left of the ".". In this way, the "." is a ***name qualification*** that means "access component".

A similar problem arises with conflicts in operation names. For example, in the Aircraft and Mission definitions above, both have an operation named "ChangeState". To avoid ambiguity in the different definitions of the ChangeState operations, one could employ the same form of name qualification as used for defining the two State objects. That is, define operation Aircraft.ChangeState and Mission.ChangeState.

Another means to qualify operations definitions is to use ***overloading***. Overloading is the means to disambiguate two operations of the same name based on the types of the inputs and outputs of the operations. Consider the following definitions for two different ChangeState functions:

```
operation ChangeState is
   components: ...
   inputs: Aircraft.State
   outputs: Aircraft.State
   description: {State change operation for Aircraft State.}
end ChangeState

operation ChangeState is
   components: ...
   inputs: Mission.State
   outputs: Mission.State
   description: {State change operation for Mission State.}
end ChangeState
```

Here, what distinguishes the two operations is not a difference in the name, but rather the fact that each operation takes a different type of input and produces a different type of output. In technical terms, the operation name "ChangeState" is ***overloaded***, since the same name is used for two different operations. In general, overloading can be used to define any two or more operations with the same name, as long as at least one input or output is different among all the definitions.

2.6.3 Names and Types

In the definitions shown thus far, the components of an entity have been shown as simple names. In the Mission object, for example, components are TaxiInOut, TakeOff, etc. A more detailed method to specify components is to use a ***name:type*** pair. For example,

```
object Mission is
    components: TIO: TaxiInOut and TO: TakeOff and etc. ...
end Mission
```

Here, the ***name*** component of the name/type pair is a local subobject name by which the component is known (the names are *TIO* and *TO* in this example). The ***type*** component is the name of some other defined object, as in the previous examples (types in this example are ***TaxiInOut*** and ***TakeOff***). Local component names have two important uses.

One use for the name/type pair notation is for reference purposes within formal requirements; this topic will be covered in a future report. The other use for name/type pairs is to enhance the expressibility of the class hierarchy. Suppose, for example, we choose to specify that all Missions have a TaxiInOut component, but it is the ***type*** of TaxiInOut that distinguishes the different special Mission instances. In this case, we could define the following form of class:

```
object class Mission is
    components: TIO:  and TO: TakeOff and etc. ...
end Mission

object NormalWeatherMission instance of Mission is
    components: TIO: NormalTaxiInOut
end NormalWeatherMission

object ColdWeatherMission instance of Mission is
    components: TIO: ColdWeatherTaxiInOut
end ColdWeatherMission
```

Here the Mission class specifies that all Missions can have a TaxiInOut phase, but that the type of this phase is left blank. The type is then specified in specific Mission instances, as in the ColdWeatherMission for example. Note that since ColdWeatherMission inherits all other components from the parent class, these need not be respecified in the instance definition. Note further that this is only an illustrative example. In this particular case, other Mission phases in addition to TaxiInOut would probably be specialized differently.

A further example that illustrates a three-level subclass hierarchy is the following:

```
object class Aircraft is
    components: Body and Engines: and WingStructure and . . .
    description: {The most generic form of aircraft.}
end Aircraft

object class JetAircraft instance of Aircraft is
    components: Engines:Jet and . . .
    description: {The primary specialization of a jet aircraft is the
        engine type.  Other specializations include ... .}
end JetAircraft

object Boeing737 instance of JetAircraft is
```

```
    components:  whatever specific components specialize this particular aircraft
    description: . . .
end Boeing737
```

2.6.4 Multiple Inheritance

It is sometimes useful to have a single instance inherit attributes from more than one parent. For example,

```
object class PassengerAircraft instance of Aircraft is
    components: Seating and . . .
end PassengerAircraft

object Boeing737P  instance of JetAircraft and PassengerAircraft is
    components: . . .
    description: {The 737 jet aircraft configured for passenger service.}
end Boeing737P
```

The rule for multiple inheritance is that the instance inherits the ***union*** of the parent attributes. In particular, if the parents have some common attributes, then the instance has only a single version of the common attributes.

2.6.5 A Final Note on Inheritance and Specialization

Instances ***inherit*** components from a parent class, can ***add*** new components to those that are inherited, and can ***specialize*** parent components that are specified with a name but without a type. Instances cannot however ***uninherit*** or ***override*** a component that is specified in the parent. Uninherit would be mean that an instance could eliminate one or more parent components. Override would mean that if a parent component were specified with both a name and a type, that the type could be changed in the instance. Not all specification languages have this ***additivity*** restriction, but ours does for simplicity.

3 Formalizing a WSRSL Specification

The previous section of the report described what amounts to the ***stylized layer*** of WSRSL. Specifying the basic objects and operations of a system provides a semi-formal definition. Fully formal definitions for objects and operations are provided in three ways:

- ***Preconditions*** and ***postconditions*** are specified for operations. The conditions are specified as predicates on inputs and outputs. They define, respectively, formal conditions that must be met before and after an operation is invoked.
- ***Equations*** are specified for objects. The equations are specified as equalities between an object's operations. They define operation relationships that formally describe an object's behavior when it is operated upon.
- ***Module axioms*** are specified as predicates over any operation within a particular specification module. They define invariants that must remain true over all operation invocations.

As described in the project proposal, our plan for the development of the stylized WSRSL is to provide a formal linkage between the stylized language level and a fully formal language level below. We have chosen Revised Special, from the EHDM system, as our underlying formal language. As our definition of the WSRSL has been refined, we have clarified the conceptual relationships between the two language levels, and improved the formal connections.

- parameterized modules
- syntactic overloading
- support for state objects and operations

In a similar vein, the stylized WSRSL provides the following syntactic and semantic enrichments over Revised Special:

- object constructor primitives
- classes and inheritance
- overall simplified syntax

The conceptual relationship between WSRSL and Revised Special is analogous to the relationship between Revised Special and multi-sorted higher-order logic (see Figure 1). In the Revised Special language definition manual [C* 86], the authors indicate that Revised Special provides a number of syntactic and semantic "enrichments" over standard higher-order logic.

In the same way that all the constructs of Revised Special can be formally represented in higher-order logic, so can all the constructs of the WSRSL be formally represented in Revised Special. Hence the WSRSL itself is founded ultimately in formal logic. This is what will allow formal verification of and reasoning about WSRSL specifications.

While the formal layer of WSRSL is based on EHDM, the syntax of the EHDM-derived constructs is part of WSRSL. That is, specifiers using WSRSL do not ever use EHDM directly, but the formal layer of WSRSL.

Due to a later-than-expected arrival of EHDM, our work on WSRSL/EHDM interface has been somewhat delayed this year. We have formulated an initial set of relationships between major WSRSL constructs and their equivalent representations in EHDM. Table 1 summarizes these relationships. A discussion of the table contents follows.

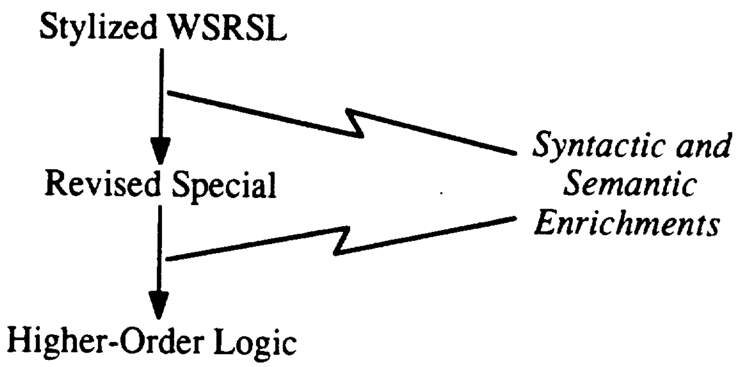


Figure 1: Relationship between WSRSL and Revised Special

WSRSL Construct	EHDM Construct
module	module
object	type
and component	selector function
or component	tagged selector function
list component	type
equations	definitional axioms
axioms	axioms
operation	function
operation signature	function signature
inputs/outputs	function arguments
pre/post conditions	hoare formulae
predicates	expressions
other attributes	comments
class/instance	suitably named instances

Table 2: WSRSL Features and EHDM Equivalents

The primary packaging construct in both WSRSL and EHDM is the module. Modules contain the major definitions in both languages.

One of the more significant enrichments of WSRSL over EHDM are the constructs for specifying object components. In EHDM, as in other similar formal specification languages, ***types*** are simply identifiers, representing arbitrary sets of objects of the same type. Other than functions defined on types, the typed objects themselves have no structure. Hence, the WSRSL object composition primitives must be represented in EHDM as explicit functions. There is in fact a reasonably standard form of composite object representation in strongly-typed formal specification languages. Viz., the components are represented by defining explicit ***selector functions***. For example, if object A has components a1:t1 and a2:t2 and a3:t3, then the formal EHDM representation defines three selector functions with signatures:

Select_ai: function [A -> ti]
for i = 1 to 3.

Selector functions for **or**-composed WSRSL objects are the same as for **and**-composed objects, with the addition of a tag function that holds the current object type. The **list**-composed WSRSL objects require no selector functions, since list composition represents the basic zero-or-more type mechanism standard in EHDM.

Object equations in WSRSL are a restricted form of axioms in EHDM. The restriction is that object equations are in the form of abstract datatype algebraic equations, which are syntactically a proper subset of EHDM axioms.

WSRSL operations and EHDM functions are essentially the same constructs, except for minor syntactic differences, and the fact that WSRSL operations can be multi-valued. Multi-valued functions are typically defined as single-valued functions that return list-valued objects, which can be readily accomplished in EHDM.

WSRSL pre/post conditions will map reasonably directly to EHDM hoare formulae. Determining the precise details of the mapping requires further study, but there are no major conceptual problems.

All predicates used in WSRSL specifications, including pre/post conditions and module invariants, are essentially the same as EHDM expressions, with only minor syntactic variations. WSRSL uses EHDM's ASCII-text representations for quantifiers and other non-ASCII logical operators. We will strive to keep WSRSL and EHDM expression syntax as close as possible, changing only where clarity at the stylized level would be compromised by using EHDM syntax directly.

As described in Section 2, WSRSL allows the definition of an arbitrary number of general-purpose attributes for objects and operations. Semantically in WSRSL, these user-defined attributes are just comments, and hence their representation in EHDM as comments is natural.

The final major enrichment of WSRSL over EHDM is the class/instance inheritance construct. Inheritance does not add any additional logical power to WSRSL, just convenience of definition. In EHDM, parameterized modules and suitable instance naming can be used to represent WSRSL-style inheritance. The precise form of the translation requires further study. We propose to complete this study, and fully define the WSRSL-to-EHDM translation in the coming year.

4 The WSRSL Methodology

The methodology described here was developed in conjunction with the analysis of ASCT for the purposes of translating it into WSRSL. The major analysis involves the codification of the objects and operations. This is essentially the "domain analysis" process described by Neighbors [Neig 84] and others, wherein domain experts identify objects and operations as the primary entities into which a requirements specification is decomposed. In the case of a requirements documentation such as ASCT, many of the objects and operations are derived directly from the dataflow diagrams.

In conjunction with object/operation decomposition, other properties of the requirements specification must be formalized using the more formal, EHDM-based layer of ASCT. Such analysis involves the translation of often inconsistent natural language statements into the target formal language. Pursuant to this translation, our methodology uses some English-to-Spec-Language transformations that guide a human analyst. We have formulated an initial list of such transformations, as discussed in the next section of the report.

One of our fundamental propositions of this project is that in the future, requirements specification should be done initially using a formalized language, not with free-form natural language as in much of ASCT. This is in fact an old proposition, e.g., from the SADT days, but a proposition that is as valid as ever.

Given for this project that we did in fact have an existing document from which to work, our methodology contains a number of steps that would not be necessary if the requirements methodology had been developed from the start in WSRSL. A summary of the specific methodology steps for analyzing ASCT is the following:

- Deducing objects and operations from DFDs
- Deducing new objects and operations from prose requirements descriptions
- Deducing object structure from prose requirements descriptions
- Deducing modules where none existed
- Deducing formal requirements from prose
- Identifying meta-requirements that specify how requirements are to be tested, validated, or verified
- Identifying auxiliary knowledge domains
- Removing superfluous prose

Starting from ASCT, the first step in identifying objects and operations is performed by inspection of the dataflow diagrams (DFDs). Basically, each DFD transform is an operation and each flow is an object. One of the fundamental deficiencies of system analysis based purely on DFDs is that object descriptions are underemphasized. That is, DFD-based analysis tends to present predominantly an operation-oriented view of a system. As pointed out in the introductory discussion in Section 2 of the report, a requirements specification should present *both* operation-oriented and object-oriented views.

In terms of analyzing ASCT, what is necessary is to deduce object structure from the text portions of the document, since it is not defined explicitly in the DFDs. Our analysis in this area revealed that a number of important objects, such as the aircraft and crew, were weakly defined in ASCT. These weaknesses have been addressed by specifying Aircraft and Crew as two important object-oriented modules in the WSRSL specification.

In addition to deducing the basic object/operation structure of ASCT, we needed to deduce an appropriate modular structure. Decomposing any specification into modules is always a partly qualitative process; there is rarely a single "best" modularization. The ASCT modules that we chose are based largely on the functional organization of the DFDs, with the addition of the above-mentioned object-oriented modules.

Once the DFDs have been analyzed and modules have been defined, the further refinement of objects and operations is carried out by analysis of the prose text. Also, the formal specification of requirements is carried out by prose analysis. The subsections that follow describe the further details of these analyses.

4.1 What is a "Requirement"

In the discussion to this point, we have not clearly delineated the distinction between "requirements" and "specification". By choosing a wide-spectrum approach to requirements specification, our position is that the distinction is not clear cut. Systems described in WSRSL contain aspects of both requirements and specification.

Given that we want working definitions for the terms, we define them as follows:

- A ***specification*** is the basic object/operation decomposition of a system.
- A ***requirement*** is some verifiable statement made about an object or operation.

As described in Section 3, there are three forms in which to express verifiable statements about an object or operation: ***operation pre- and post-conditions***, ***object equations***, and ***module axioms***. Hence, in WSRSL, a "requirement" is formally one of these three forms of statement.

From a practical standpoint, this distinction is not all that useful, particular when translating a prose document such as ASCT. In almost all cases, ASCT "requirements" are expressed as a combination of object/operation descriptions and formal statements. The point is that unless one has defined an entity to which a "requirement" refers, the "requirement" cannot be stated.

A tenet of the WSRSL methodology is that requirements should be developed in stylized form first, followed by formalization. In the WSRSL translation of ASCT, this is accomplished by defining operation attributes that contain the informal prose description of requirements. For example, operation fields such as "CMF" and "CAB" are used to state the informal descriptions of requirements for "Control Mission Flight" and "Control Aerodynamic Braking." Additional comments surrounding the formal statement of the requirements link to the informal description. In addition, the hypertext linking facility of the WSRSL browser is used to define these links formally. This form of linking is described in detail in Section 5 below.

4.2 A Detailed Formalization Example

This subsection presents a sample of the kind of formalization that must be conducted in order to translate requirements such as they are stated in ASCT into a formalized form. As a relatively self-contained example, we present the formalization for the Spiral Mode requirements stated on page 103 of ASCT. First the requirements are reprinted verbatim as they appear in ASCT. Then a set of transformation guidelines is presented. Finally, the requirements are stated in fully formal form, as they would appear in the context of a complete WSRSL requirements specification document.

4.2.1 Verbatim Excerpt from the ASCT Document

```
Spiral Mode (U.A.S.4)

a) If unstable, the spiral mode time to double amplitude shall be no less
than 20 seconds at speeds  from 1.2 VS1 to VFC/MFC.  (BCAR, D2-8,2 -
Conventional Control)

b) The airplane characteristics shall not exhibit coupled roll-spiral mode in
response to the pilot roll commands.  (MIL-F-8785C 3.3.1.4 - Conventional
Control)

c) Minimum acceptable: the spiral mode time to double amplitude shall be
greater than 4 seconds  (MIL -F08785C 3.3.1.3 - Conventional Control)
```

4.2.2 Translation Rules

The semi-formal English prose used to state requirements in this document excerpt is typical of that found throughout ASCT. In formalizing such requirements, a number of common transformation rules are needed. It is our intention to categorize such rules for the purposes of an organized translation process. However, as stated in the original proposal, automation of such transformation rules is well beyond the scope of this project.

For the task at hand, viz., translating ASCT, the benefit of a transformation catalog can be great, given that many of the transformations are reusable. Ultimately, however, the use of such catalogs should diminish, if not disappear altogether, since requirements specifications will not be developed in a form such as ASCT, but rather developed initially using the WSRSL, eliminating the need for the style of ASCT translation we are now conducting.

Excerpt-Specific Transformation Rules

"unstable" ==> ***plane.stable* = *false***
"at all speeds from 1.2 VS1 to VFC/MVC." ==> $\forall$ ***vel: 1.2VS1*** $\leq$ ***vel*** $\leq$ ***VFC/MFC***
"airplane characteristics" ==> ***plane.state***
"coupled" ==> a global state var or a component of plane.state
"pilot roll control commands" ==> a class or module of commands
"the spiral mode time to double amplitude" ==>

if Aircraft.State.Mode = "Spiral" and Aircraft.State.Time = t and
Aircraft.State.Amplitude = a then
some quantification over t : Aircraft.State.Amplitude ***rel*** 2 * a

General Natural Language Transformation Rules

"at <range>" ==> $\forall$ ***x in <range>***
"in response to" ==> a postcondition of some operation invoked in the same sentence
"minimum acceptable" ==> typically a no-op
"shall", "must" ==> a condition must obtain (i.e., become true)
"shall with probability" ==> a condition must obtain with given probability

"should" ==> a condition will obtain if it does not negate some "shall"
"exhibit" ==> =
"command" ==> ***operation***
"commands" ==> ***operation class***
"coupled (with)" ==> ***and*** (context dependent)

4.2.3 Formal Translations of Requirements

Given the above translation rules, the following are the resultant fully formal statements of the three "Spiral Mode" requirements:

```
a) if Aircraft.State = Unstable then
      if Aircraft.State.Mode = ''Spiral" and Aircraft.State.Time = t and
          Aircraft.State.Amplitude = a and
          1.2 * VS1 <= Aircraft.State.Speed <= VFC/MFC then
        exists t <= t1 <= t+20 : Aircraft.State.Amplitude = 2 * a

b) module PilotCommands
     operation RollControl
       postcondition: Aircraft.State.Mode ~= ''CoupledRollSpiral"
         . . .
     end RollControl

c) forall s in Aircraft.State :
     if s.Mode = ''Spiral" and s.Time = t and s.Amplitude = a
       forall t <= t1 <= t+4 :
         if s.Time = t1 then s.Amplitude < 2 * a
```

4.3 Rationale for the Use of a Wide-Spectrum Language

In the original project proposal, we indicated that our requirements specification methodology would be divided into four phases: Stylized Requirements Analysis, Stylized Specification, Formal Requirements Analysis, and Formal Specification. In the course of our ASCT analysis, we have come to believe that these phases should be more tightly integrated than originally planned. In particular, rather than have four distinct tiers of stylized and formal requirements and specification languages, our initial work has led us to the development of the single ***wide spectrum*** requirements/specification language that allows the analyst to gracefully refine requirements into specifications and to gracefully add formal components to an initially informal document. The notion of wide spectrum languages has been gaining favor among practitioners and researchers in software engineering.

The use of a wide spectrum language does not mean that we will abandon the notion of the separate phases, just that the vehicle for expressing the phases allows inter-phase integration to be readily accomplished. Different audiences of a full wide-spectrum document will view it through different ***filters***. For example, a high-level management audience will view only those aspects of the document that are appropriate to the management perspective.

4.4 Auxiliary Knowledge Domains

At the outset of this project, we recognized the need for the representation of domain knowledge in a formal requirements specification. We have subsequently observed that in addition to specific domain knowledge, such as is required to analyze aircraft missions, we need to represent ***auxiliary knowledge domains***. Such domains should be organized in a modular form, consistent with the other portions of the requirements specification document.

Two important auxiliary knowledge domains that we have so far identified are ***Timing*** and ***Statistics***. The former is used to formally specify system timing requirements. The latter is necessary to formally specify requirements of a probabilistic or statistical nature. It is ***not*** the intent to specify such modules fully for this project. We will identify the auxiliary knowledge modules necessary to obtain a fully formal result and include a small number of objects and operations that would inhabit such modules. Detailed specification of such modules is beyond the scope of this project.

5 WSRSL Prototype Tools

An important adjunct to any formal language is a computer-based environment. Researchers in software engineering have recognized this fact for a number of years. Generally speaking, a language environment includes a number of tools to edit and analyze documents expressed in the subject language.

Figure 2 depicts the components of an overall environment for WSRSL. During this year of the project, we have built prototypes for three of these environment components: the Parser, Browser, and Dataflow Editor. These prototypes are described in the following three subsections. In the coming year, we propose to develop the other two environment components: the Interface Generator and the Verifier.

5.1 WSRSL Parser

In order for WSRSL specifications to be formally checked, a computer-based language translator, i.e., a parser, must be developed. Our prototype parser has the following features:

- performs complete syntactic analysis
- performs specification type checking to ensure syntactically complete and consistent definitions
- is integrated within the framework of a general-purpose system development environment called *Blend*

The syntactic analysis and type checking features of the WSRSL parser are critical to ensuring correct specifications. Further, the syntactic analyzer produces a formal internal form for a specification that can be more readily mapped to the underlying Revised Special level.

Integrating the parser within the Blend environment makes available a number of environment tools that has facilitated the development of the WSRSL browser described in the next section of the report. Blend integration will also facilitate the development of specification traceability tools, through which the objects and operations at the requirements specification level can be traced to their corresponding data and functions at the design and implementation levels. We hope to develop such additional tools in the future.

5.2 WSRSL Browser

The major features of the WSRSL browser are:

- Multi-window, menu-based interface (running under X Windows)
- Mixed text and graphical browsing capabilities
- Generalized hyper-object navigation

A sample screen for the top-level browser window is shown in Figure 3. This level of the browser provides an overall "roadmap" for the specification. The screen is divided into two panes. The Left pane contains an object roadmap and the right pane an operation roadmap.

The roadmaps show the hierarchical structure of each of the entities defined in one or more specification modules. The module boundaries are shown as dashed boxes surrounding the objects and operations contained in the modules.

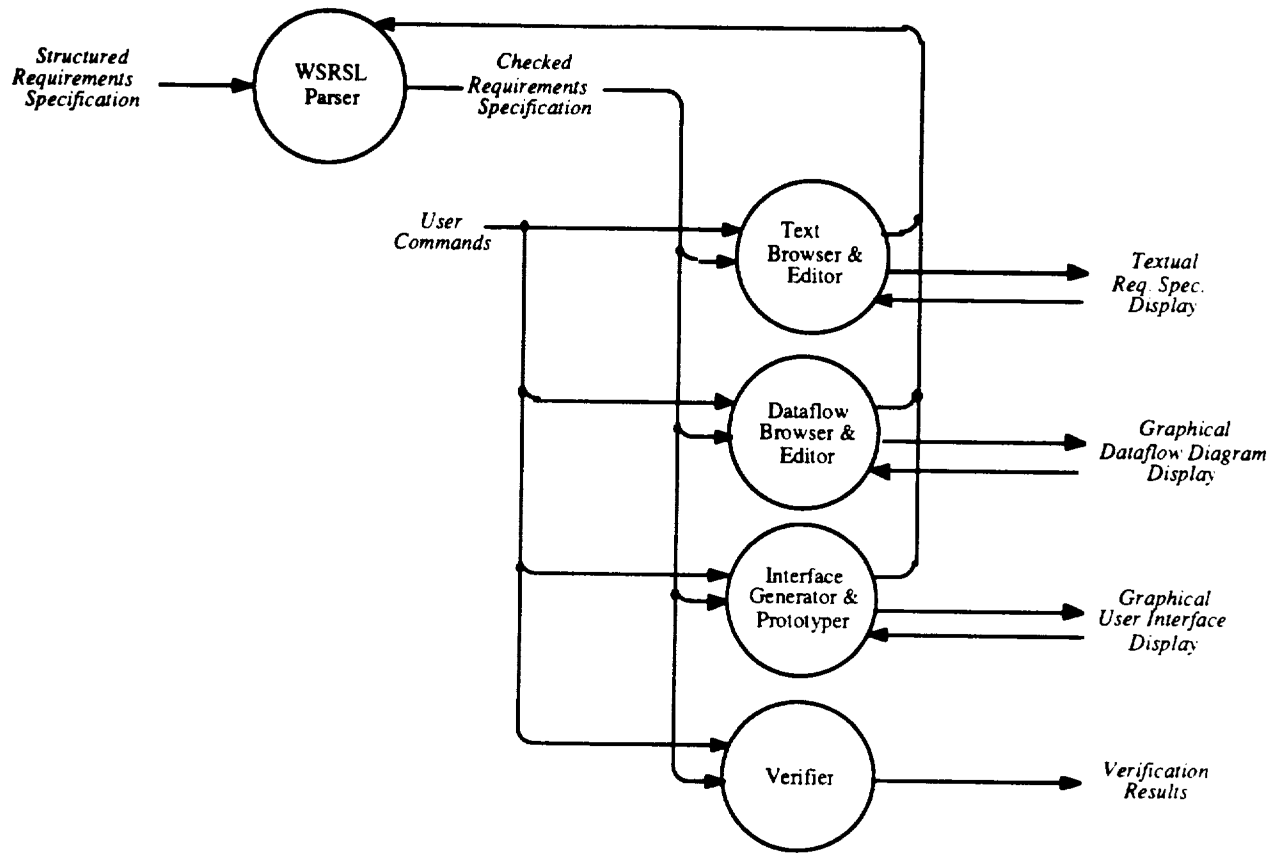

Structured Requirements Specification
WSRSL Parser
Checked Requirements Specification
User Commands
Text Browser & Editor
Textual Req. Spec. Display
Dataflow Browser & Editor
Graphical Dataflow Diagram Display
Interface Generator & Prototyper
Graphical User Interface Display
Verifier
Verification Results

Figure 2: WSRSL Environment Components

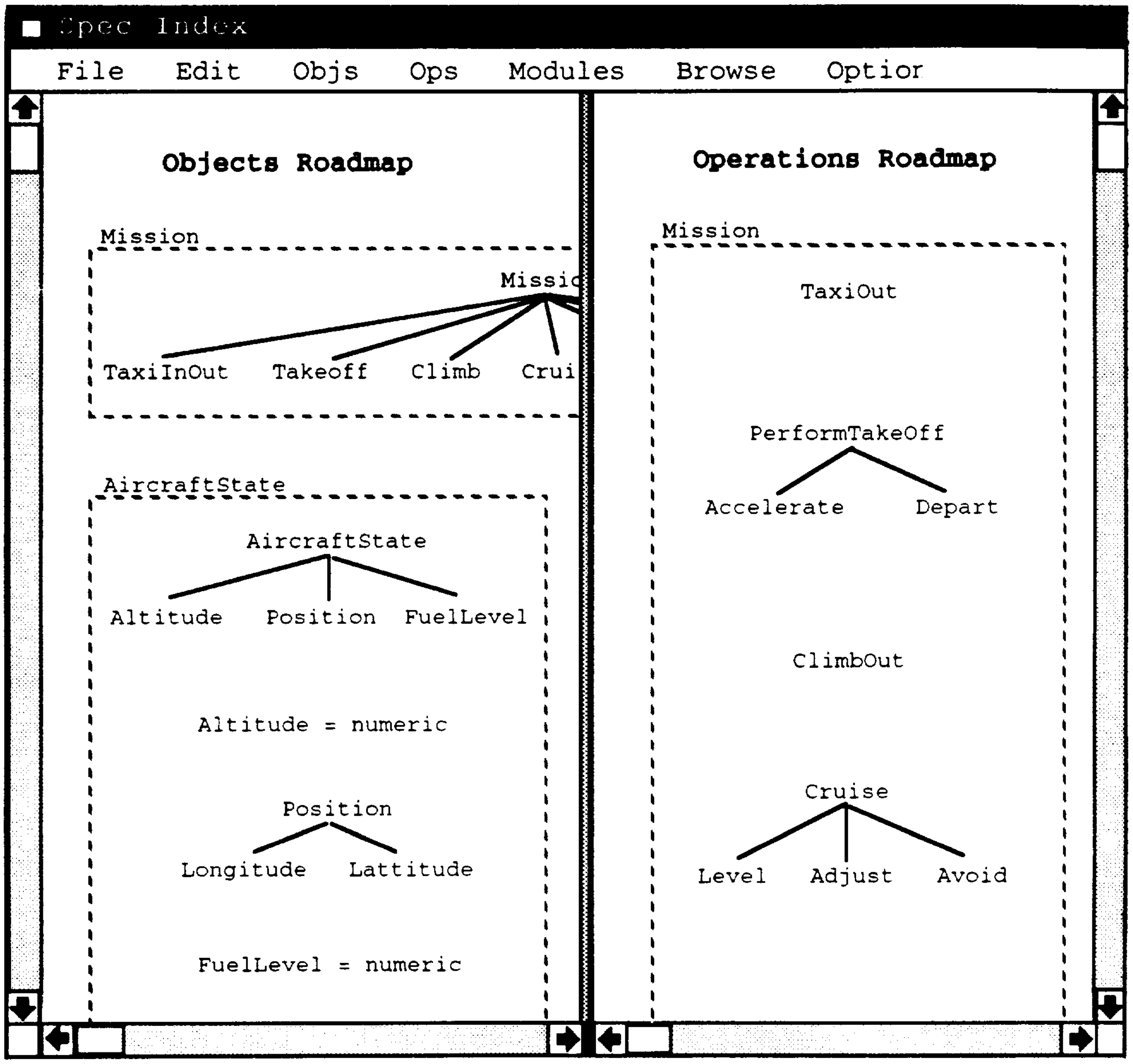
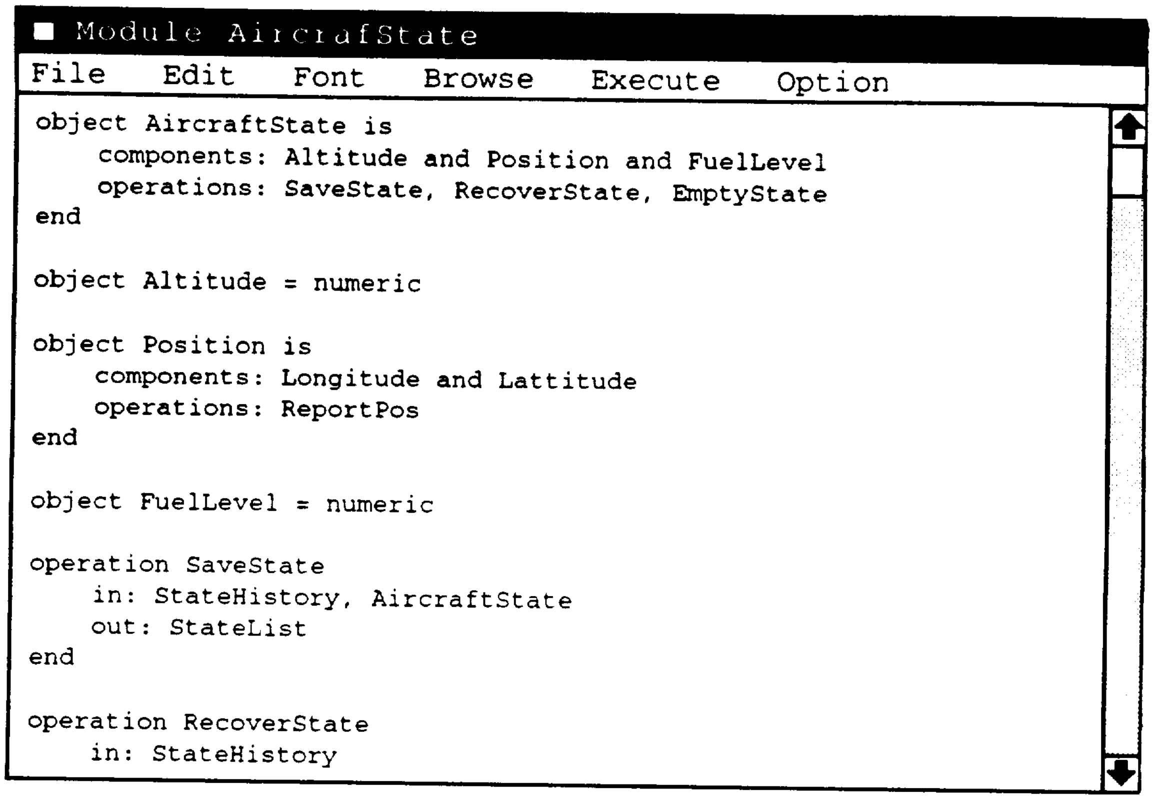


Figure 3: Top-Level Browser



```
object AircraftState is
    components: Altitude and Position and FuelLevel
    operations: SaveState, RecoverState, EmptyState
end

object Altitude = numeric

object Position is
    components: Longitude and Lattitude
    operations: ReportPos
end

object FuelLevel = numeric

operation SaveState
    in: StateHistory, AircraftState
    out: StateList
end

operation RecoverState
    in: StateHistory
```

Figure 4: Module Text Browser

Since a specification may contain a large number of entity definitions, the browser provides the means to conveniently navigate through a large collection of definitions. First, the object and operation roadmap windows are scrollable in all directions, allowing the user to focus on specific entity definitions. Three menu entries on the top of the browser window allow navigation by selecting an entity by name:

- The ***Modules*** menu contains a checklist of modules that are currently loaded
- The ***Objs*** menu provides an alphabetic list of objects by name
- The ***Ops*** menu provides an alphabetic list of operations by name

Checking a module name in the ***Modules*** list causes the names of all the module's objects and operations to appear on the ***Objs*** and ***Ops*** lists. Selectively checking and unchecking names on the ***Modules*** list allows the length of the ***Objs*** and ***Ops*** lists to be controlled. (Clearly, an alphabetic list of hundreds or even thousands of entity names would be impractical to use.) Selecting a name from the ***Objs*** or ***Ops*** lists causes the roadmap display to scroll to the selected name.

Figure 4 shows a sample module text browser. This form of browser window presents the next level of detail in a specification. The window contains the scrollable text of the full module specification in the WSRSL. A textual module browser appears when the user selects a specific module in one of the roadmap windows and executes the 'ZoomIn' command on the top-level ***Browse*** menu selection. Within a text browser, users may navigate by direct scrolling, or searching for specific entity definitions by name.

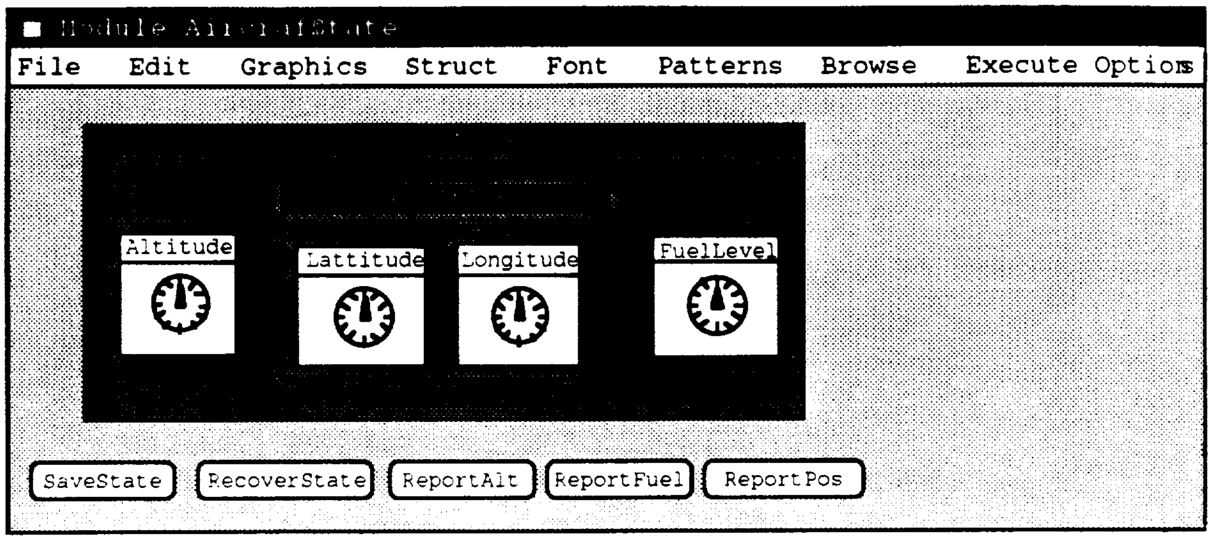


Figure 5: Module Graphic Browser

Figure 5 shows a sample module graphic browser. Figure 5 shows the same module as shown in Figure 4, but here in graphical form. Each type of object and operation have a standard graphical view. Hierarchical object nesting is shown as graphically nested objects. Operations are shown as labeled buttons.

An important feature of the browser is the linkage of the textual and graphical views. Both views present exactly the same module structure, just in different physical forms. Further, editing either view automatically updates the other. This strong connection is made possible by the integration of the WSRSL language translator within the framework of the Blend environment, as described earlier in the report.

At the current stage of environment development, the text browser is a usable prototype. The graphical views have only been prototyped in very preliminary form. We hope to continue refinement of the text browser, and development of the graphical browser in the coming year.

The WSRSL browser has a ***hypertext linking*** feature. Hypertext browsing allows "electronic footnotes" to appear in documents. To follow a footnote, the user clicks a noted spot on the screen, whereupon a new window appears with the footnote information. Hyper-links are more general than text-based footnotes in that any text or graphic object appearing in a window can have one or more links to one or more other objects. The WSRSL hypertext features are comparable to similar features found in other hypertext systems, including Apple's HyperCard.

Browser users can create and follow hyperlinks by selecting from the Browse menu, which has the following entries:

- Create Link
- Follow Link
- Show Links
- Remove Link

The 'Create Link' selection prompts the user with the dialogue box shown in Figure 6

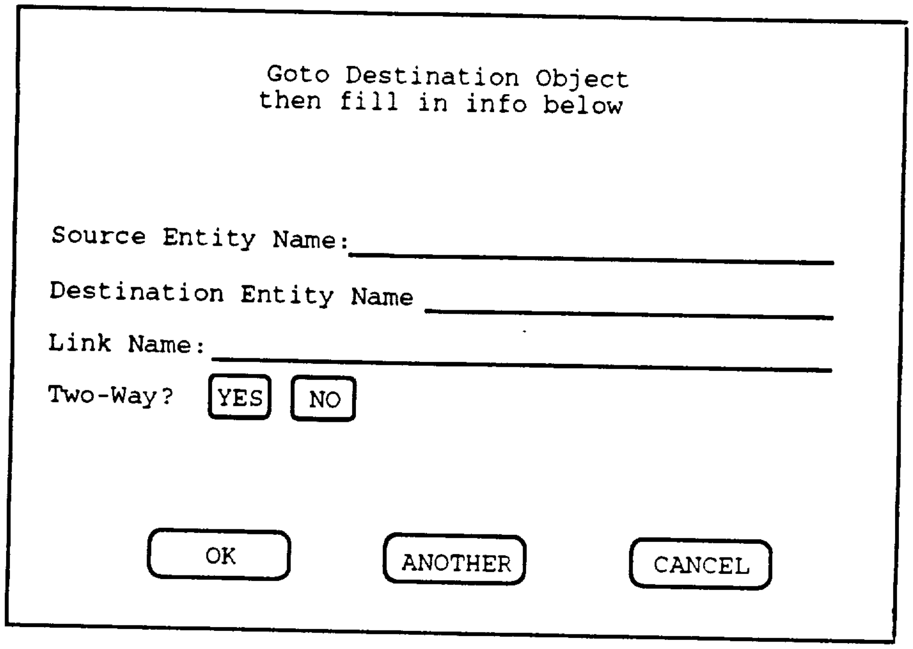

Goto Destination Object
then fill in info below

Figure 6: Creating a Hyperlink

The 'Follow Link' selection provides a menu of all linked entities available from current screen context, and allows the user to select the link to follow. When the 'Follow Link' selection is made, the screen is updated to contain the definition of the entity on the end of the followed link. The 'Show Links' selection highlights all objects in current window from which links emanate. It is noted that many-to-one and one-to-many links are definable, providing a very general navigation facility.

A number of basic link types are built-in to the browser, based on the hierarchical structure of a specification and the temporal browsing history. These built-in links are:

- **Next** – goto the next entity defined in the module
- **Prev** – goto the previous entity defined in the module
- **First** – goto the first entity defined in the module
- **Last** – goto the last entity defined in the module
- **Up**... – goto the parent entity of which the current entity is component
- **Down**... – goto the first child (i.e., component) of the current entity
- **Back** – goto previously browsed node
- **Root** – goto top-level module
- **Goto**... – goto a specific entity by name

5.3 Dataflow Tool

The dataflow browser includes features now common in several software engineering tools, including the one used to develop the original ASCT dataflow diagrams (DFDs). These features include:

- basic DFD node/arc creation
- node/arc labeling
- arc rubber banding when nodes are moved
- diagram leveling
- ability to save/load a textual version of a displayed DFD

The major improvement of our DFD browser over other comparable tools will be its full integration in the WSRSL environment. This will allow requirements specifications to be viewed in textual form, the ***structural*** graphical form described in last quarter's report, and the ***functional*** graphical form of a DFD. Integration of the DFD tool with the WSRSL browser will also mean that hyper-object navigation features will be available during DFD browsing.

References

[CM 91] G. C. Cohen, R. E. McLees, "An Example of Requirements for Advanced Subsonic Civil Transport (ASCT) Flight Control System Using Structured Techniques," NASA Contractor Report 187256, 1991.

[C* 86] J. S. Crow, et al., "SRI Specification and Verification System, Preliminary Definition of the Revised Special Specification Language, Version 3.0," SRI Project 5725, 75 pp., May 1986.

[GMB 82] S. J. Greenspan, J. Mylopoulos, and A. Borgida, "Capturing More World Knowledge in the Requirements Specification," **Proceedings of the Sixth International Conference on Software Engineering**, pp. 225-234, 1982.

[Neig 84] J. M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components" **IEEE Transactions On Software Engineering, SE-10,5**, pp. 564-574, September 1984.

[RS 77] D. T. Ross, K. E. Schoman, "Structured Analysis for Requirements Definition," **IEEE Transactions on Software Engineering, SE-3, 1**, pp. 6-15, January 1977.

[TH 77] D. Teichroew, E. A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," **IEEE Transactions on Software Engineering, SE-3, 1**, pp. 41-48, January 1977.

[YC 79] E. Yourdon, L. L. Constantine, **Structured Design**, Prentice-Hall, 1979.

A WSRSL Syntax

```
<spec> → spec <spec name> <entity spec list> <formal part> end <spec name ender> ';'
<entity spec list> → { <entity spec> ';' ... }
<entity spec> → <object spec> | <operation spec>
<object spec> → <obj symbol> [class] <obj name> [instance] is <obj body> end <obj name ender
<operation spec> → <op symbol> [class] <op name> [instance] is <op body> end <op name ender
instance → instance of [<class name> ...]
<obj body> → '=' <obj expr> | components ops { <obj attributes> ';' ... }
<op body> → components inputs outputs { <op attributes> ';' ... }
<components> → <components symbol> ':' <components spec>
<components spec> →
        empty |
        <components spec> <comp op> <components spec> |
        <prefix list op> <components spec> |
        <components spec> <postfix list op> |
        <name type pair> |
        name |
        '(' <components spec> ')'
<comp op> → <and op> | <or op>
<and op> → and | ','
<or op> → or | '|'
<prefix list op> → list | list of | collection | collection of
<postfix list op> → '*'
<ops> → <ops symbol> ':' [<op signature> , ...]
<inputs> → <in symbol> ':' { <init name type pair> , ... }
<outputs> → <out symbol> ':' { <init name type pair> , ... }
<op signature> → <op name> ':' <op parms>
<op parms> → empty | {<obj name> ',' ... } '– >' {<obj name> ',' ...}
<obj attribute> → id ':' comment
<obj attribute> → id ':' comment
<name type pair> → name ':' <obj name>
<init name type pair> → <name type pair> |
        <name type pair> ':=' <obj expr>
<obj expr> → <obj atom> | '[' <obj expr> ',' <obj atom> ']'
<obj atom> → string | integer | number | boolean | nil <var name>
<formal part> → vars conditions equations
<vars> → { <var decl> ';' ... }
<var decl> →<ar symbol> <var name list> ':' <obj name>
<conditions> → conditions { <pre post conds> ';' ... }
<pre post conds> → <op symbol> <op signature> precond postcond
<precond> → <pre symbol> ':' predicate
<postcond> → <post symbol> ':' predicate
<predicate> →
        predicate <pred bin op> <rel expr> |
          <pred pre op> predicate |
          if predicate then predicate |
          if predicate then predicate else predicate |
          forall <var name list> ':' predicate |
          there exists <var name list> ':' predicate |
          exists <var name list> ':' predicate |
          <rel expr>
<rel expr> →
          <rel expr> <rel bin op> <arith expr> |
          <arith expr>
<arith expr> →
          <arith expr> <arith bin op> <functional expr> |
          <arith pre op> <arith expr> |
          <functional expr>
<functional expr> →
          <var name> |
          <op name> '(' { predicate ',' ... } ')' |
          '(' predicate ')'
<pred bin op> → and | or | in | implies
<pred pre op> → not
<rel bin op> → '=' | '<' | '>' | ' =' | '<=' | '>=' | in
<arith bin op> → '+' | '-' | '*' | '/' | div | mod
<arith pre op> → '+' | '-'
<equations> → { equation ';' ... }
<equation> → <eq symbol> lhs '==' rhs
<lhs> → <functional expr>
<rhs> → predicate

<name> → identifier
<spec name> → identifier
<spec name ender> → empty | identifier
<class name> → identifier
<obj name> → identifier
<obj name ender> → empty | identifier
<op name> → identifier
<op name ender> → empty |identifier
<var name> → identifier
<obj symbol> → object | obj
<op symbol> → operation | op
<components symbol> → components | components
<ops symbol> → operations | ops
<in symbol> → inputs | in
<out symbol> → outputs | out
<var symbol> → variable | var
<eq symbol> → equation | eq
```

<pre symbol> → **precondition** | **pre**
<post symbol> → **postcondition** | **post**

<identifier> → <letter> { <letter> | <digit> ... }
<string> → '"' { <any character> ... } '"'
<integer> → { digit ... }
<number> → integer '.' integer
<boolean> → **true** | **false**
<comment> → '{' { <any character> ... } '}'
<any character> → ***any legal lexical character***
<empty> →

B Excerpts of ASCT in WSRSL

```
{*****************************************************************************
 * FlyMission is the top-level module of the system.  It is derived from the
 * material on ASCT Pages 9-15.
 *****************************************************************************}
module FlyMission is  { from pg 13 }

  import Navigate, ControlMissionFlight;
  export Mission, TargetFlightPath, ActualFlightPath, ExternalForcesOnActuator;

  object Mission is
    components: TaxiInOut and TakeOff and Climb and Cruise
            and Descent and Approach and Landing and AltitudeRange and State
    operations:
      Navigate: Mission -> TargetFlightPlan
      { There should be additional operatins that are not explicitly specified
      in ASCT.}
    description:
      { Definition of particular flight mission from which the target flight
      path can be generated (ASCT pg. 13).  A Mission is the main object of the
      ASCT Flight Control System.  Its first seven components represent each of
      the main segments of a controlled flight (ASCT pg. 15).  The last two
      components represent the altitudes that may be attained during a mission
      (from 0 to MaxAltitude) and the global states of the mission.}
  end Mission;

  object Mission.State is
    components:
      OP: OperatingProcedures;
      FP: FlightPlan;
      FE: FlightEnvelope;
    operations: ;
    description: ;
  end Mission.State;

  {**** Mission Segments (pg. 15) ****}

  {** These three field definitions are from Table 1 on page 15. **}
  object field control_action;
  object field driver;
  object field control_system_requirement;

  object class MissionSegment is
    components: AltitudeRange;
  description:
    { A generic mission phase.  Only identified component from ASCT is
    altitude range, but presumably there should be more. };
end MissionSegment;

object TaxiInTaxiOut instance of MissionSegment is
  components:  MoveFromTerminalPhase and AltitudeRange;
  operations: ;
  control_action: { Move from passenger terminal to runway. };
  driver: { Terrain and obstacle avoidance. };
  control_system_requirement: {Speed control, nosewheel steering.};
end TaxiInTaxiOut;

object TakeOff instance of MissionSegment is
  components: RunwayAcceleration and RunwayDeparture and AltitudeRange
  operations:
      AccelerateToTakeOff:  AircraftState, SpeedControls -> AircraftState
      DepartRunway: AircraftState, LiftOffControls -> AircraftState;
  control_action:
    { Accelerate to takeoff speed and depart runway. }
  driver:
    { Runway length, thrust limits, crosswind conditions };
  control_system_requirement:
    { Set height lift, set takeoff trim, thrust setting, nosewheel steeriing,
    engine out augmentation, on ground braking, stall angle of atack warning,
    manual trajectory control };
end TakeOff;

object ClimbOutAndClimb instance of MissionSegment is
  components: ClimbOut and ClimbToAltitude;
  control_action:
    { Ascend to cruise altitude< and speed.}
  driver:
    { Time constraint, fuel consumption, ease pilot workload, ride quality };
  control_system_requirement:
    { Thrust setting, manual trajectory control, auto trajectory control,
    manual and auto trim envelope protection, auto control limiting, lift
    config. };
end ClimbOutAndClimb;

object Cruise instance of MissionSegment is
  components: ;
  operations: ;
  control_action:
    { Cruise. };
    driver:
      { Ease pilot workload, fuel consumption, minimize drag, ride quality. };
    control_system_requirement:
      { Speed control, manual trajectory control, auto trajectory control,
      manual an auto trim, envelope protection, auto control limiting, lift
      control. };
  end Cruise;

  object DescentAndApproach instance of MissionSegment is
    components: Descent and Approach;
    control_action:
      { };
    driver:
      { Ease pilot workload, ride quality, crosswind conditions, all weather
      approaches, tight path following. };
    control_system_requirement: { };
      { Speed control, manual trajectory control, auto trajectory control,
      manual and auto trim, envelope protection, auto control limiting, lift
      control. };
  end DescentAndApproach;

  object Landing instance of MissionSegment is
    components: Deceleration and Touchdown ;
    control_action:
      { Flare, touchdown and decelerate to taxi speed. };
    driver:
      { Runway length, crosswind conditions, rapid speed change, tight path
      following all weather landings, ease pilot workload. };
    control_system_requirement:
      { Speed control, manual trajectory control, auto trajectory control,
      envelope protection, auto control limiting, lift control, stall angle of
      atack warning. };
  end Landing;

  object MissedApproach instance of MissionSegment is
    control_action:
    driver:
      { Rapid thrust change; quick, hard maneuvers. };
    control_system_requirement:
      { Terrain and obstacle avoidance, wind shears, ride quality. };
    description: ;
      { Thrust control, manual trajectory control, envelope protection, lift
      control, engine out augmentation, stall angle of attack };
  end MissedApproach;
```

```
object class FlightPath is
  components: Direction, Angle, ... ;
  operations: ;
  description: ;
end FlightPath;

object ActualFlightPath instance of FlightPath is
  components:  {Inherited from FlightPath.}
  description:
    { The sensed 4 dimensional flight path and attitudes of the aircraft as
    well as any other sensed values necessary to satisfy the control
    requirements. (See page 13.) }
end ActualFlightPath ;

object TargetFlightPath instance of FlightPath is
  components:
  description:
    { The desired 4 dimensional flight path and attitudes generated by some
    navigation function.  (See page 13.) }
end TargetFlightPath ;

object AircraftAttitudes is
  components: Pitch, Roll, Heading;
  description:
    { Aircraft pitch, roll and heading attitudes.  (See page 13.) }
end AircraftAttitudes;

object TargetFlightPath is { from pg 13 }
  components:
  operations:
  description:
    { The desired 4 dimensional flight path and attitudes generated by some
    navigation function. (See page 13.) }
end TargetFlightPath;

 {** External forces object.  Referenced in ASCT, but not thoroughly defined.
     See translation notes for further discussion. **}
 object class ExternalForcesOnActuator is
   components: ;
   operations: ;
   description: ;
    { All forces (in particular environmental forces) other than the
    actuation forces acting on the aerodynamic braking and roll actuation
    system.};
    end ExternalForcesOnActuator;

   let t: Time;
       m: Mission;
       a: Aircraft
       fl: FailureLevel
   axiom
     if (exists t,m,a : m.Time = t and a.State.HandlingQuality = Degraded)
     then m.State = IsDegraded(m)
     endif;

   axiom
     if (forall fl,a : if Probability(fl) < 1.0*10**-9
                    and a.State.Mode = CoreControl)
     then MinimumAugmentation(a)
     endif;


  { An "external forces" axiom that states that external forces exist that
    cause anomalous conditions to arise, e.g., degraded handling quality. }
  let ef: ExternalForce
  axiom
      exists (ef : (exists t : if m.Time = t then a.State.HandlingQuality = Degrade

  aux operation MinimumAugmentation is
    inputs: A: Aircraft;
    outputs: ;
    body:
    description: ;
  end MinimumAugmentation;


  C.M.F.2
  operation EvaluateHandlingQualities is
    components: ;
    inputs: a: Aircraft, m: Mission;
    outputs: PilotRating;
    agent: Pilot
    precond: m.State.FlightEnvelope = Normal
    postcond:
    description: ;

  end EvaluateHandlingQualities;

end FlyMission;
```

```
{****************************************************************************
 * Module Crew contains material gleaned from throughout the ASCT
 * specification.  Pp. 196-196 contain very brief object descriptions of the
 * Crew, but no details.
 ****************************************************************************}
module Crew

  object class CrewMember is
    components: Name, SkillLevel, StrengthLevel;
    description:
      { Class of crew members };
  end CrewMember;

  object Pilot instance of CrewMember is
    components: PilotClassification;
    description:
      { The pilot of the mission };
  end Pilot;

  object CoPilot instance of CrewMember is
    components: PilotClassification;
    description:
      { The copilot of the mission };
  end Pilot;

  object SkillLevel is number;
  object StrengthLevel is number;

  object MissionControlSystem is
    components: ... ;
    description:
      { The onboard computer support system.  Used as agent of operation where
      appropriate (i.e., in operations that are performed automatically versus
      manually). };
  end MissionControlSystem;

  operation PerformPilotFunctions is
    components:
    inputs: PFPCFF: PilotFlightPathCommandFeelForce;
    outputs: PLTF: PilotLongitudinalTrimForce,
      PFPCF: PilotFlightPathCommandForce;
    description:
      { The functions performed by the pilot. };
  end operation AEPilot;
  operation PerformCopilotFunctions is
     components:
     inputs: CopilotFlightPathCommandFeelForce, ... ;
     outputs: CopilotLongitudinalTrimForce, CopilotFlightPathCommandForce, ... ;
      { The functions performed by the copilot. };
  end operation AECopilot;

end Crew;
```

```
{***************************************************************************
 * Module Aircraft contains material gleaned from throughout the ASCT
 * specification.  Pp. 194-197 contain brief object descriptions of the
 * Aircraft, but few details.
 ***************************************************************************}
module Aircraft

  object Aircraft is
    components: State, Structure, MajorSystems, Attitudes, { ... };
    operations: ;
    description: ;
  end Aircraft;

  object Aircraft.State
    components:
      MCM: ManualControlMode,     { The two modes of aircraft control, q.v. }
      HQ: HandlingQuality,
      NWP: NoseWheelPosition
        { . . . }
    operations: { Many. }
    description:
      { The top-level repository for all aircraft state information.  Note that
      any explicit definition of this object is conspicuously missing from
      ASCT. }
  end Aircraft.State;

  object AircraftState = Aircraft.State;

  object NosewheelPosition is
    components: ;
    operations: ;
    description:
      { Angular position of the nosewheel used for on ground low speed heading
      control. };
  end NosewheelPosition;

  object Aircraft.Structure is
    components: Engine*, EngineSupport*, PropellerShaft*, HighLiftDevices, ... ;
    description:
      { The structural components of the aircraft.  Note that only those
      components that appear in the requirements are listed here.  A full
      structural decomposition of the aircraft should be done in a complete
      structures module, and would of course be very detailed. };
  end Aircraft.Structure;
```

```
object class StructuralElement is
  components: HowMounted, WhereMounted;
  operations: ;
  description: ;
end StructuralElement;

object Engine instance of StructuralElement is
  components: Thrust... ;
  description:
    { The aircraft engine };
end Engine;

object Engine.Thrust is  {** See ASCT pg. 88. **}
  components: ;
  operations: ;
  description:
    { Thrust measurement;
end Engine.Thrust;

obj EngineThrust = Engine.Thrust; { Simple naming macro for ASCT consistency. }

object HighLiftDevices is
  components: LeadingEdgeFlap*, TrailingEdgeFlap*;
  description: ;
end HighLiftDevices;


object HowMounted is
  components: Location, ...;
  operations: ;
  description: ;
end HowMounted;

object HowMounted.Location is
  components: External or Internal;
end HowMounted.Location;

object External = ''External";
object Internal = ''Internal";


{ Major Aircraft Systems  (ASCT pg. 196 and elsewhere).  In a full spec,
 each component here should undoubtedly be represented in a separate module.}

object MajorSystems is
```

```
  components: SensorSystem, PilotControlSystem, PropulsionSystem,
    AirframeSystem, AutoFlightSystem;
  operations: ;
  description: ;
end MajorSystems

{Note that this should certainly be integrated with as a MajorSystems
 component, but it appears as an isolated object in ASCT.}
object Autopilot is
  components: ;
  description:
    { The autopilot control unit. }
end Autopilot;

{ Flight Modes and Commands }

object class Mode is
  components: ;
  operations: ;
  description:
    { A generic flight mode; specializations follow. };
end Mode

object class Command is
  components: ;
  operations: ;
  description:
    { A generic flight command.  Note that the component structure of a
    command is is not precisely clear from the various appearances of the
    term ''command" throughout ASCT.  This should be corrected. }
end Command


{ Mode Specializations }
object ManualFlightMode instance of Mode is
  components: Angle;
  operations: ;
  description:
    { Appears on ASCT pg. 129; no textual description given. }
end ManualFlightMode;

object AutoFlightMode is
  components: ;
  operations: ;
  description: ;
```

```
      { Appears on ASCT pg. 129; no textual description given. }
  end AutoFlightMode


  { Command Specializations }
  object ManualFlightPathCommand instance of Command is
    components: Angle;
    operations: ;
    description:
      { Flight path angle command generated manually (i.e., by the crew) };
  end ManualFlightPathCommand

  object AutolFlightPathCommand instance of Command is
    components: Angle;
    operations: ;
    description:
      { Flight path command generated in an automated fashion (i.e., by a
      computer system) };
  end AutoFlightPathCommand

end Aircraft;
```

```
{*****************************************************************************
 * Module Navigate is largely a place holder for information that is outside of
 * the specific focus of this document, but which should be represented
 * formally in some form in a complete document.  Pages 12 and 13 are the only
 * explicit mention of a Navigate function in ASCT.
 *****************************************************************************}
module Navigate

  { Evidently outside of the scope of this spec }
  operation Navigate is
    components: ;
    inputs: Mission;
    outputs: TargetFlightPath;
    description:
      { Generates the target flight path based on the particular mission
      requirements and anticipated and sensed environmental conditions. };
  end Navigate;

end Navigate;
```

```
{*****************************************************************************
 * Module ControlMissionFlight is the contains the top-level functional
 * components of the system, defined on pp. 17-89 of ASCT.
 *****************************************************************************}
module ControlMissionFlight

  operation field CMF;

  from FlyMission import ActualFlightPath, TargetFlightPath;
  from Mission import State;
  from Aircraft import State;

  operation ControlMissionFlight is  { pp. 13, 87 }
    components: ControlThrust, ControlPitch, ControlRoll, ControlYaw,
      ControlHeadingOnGround, ControlAerodynamicBraking, ControlBrakingOnGround,
      ControlLiftConfig, UpdateAircraftState;
    inputs: TargetFlightPath, ExternalForcesOnActuator,
      ExternalForcesOnYawActuator, EnginesThrust;
    outputs:  DisplayedLongitudinalTrimPosition, StallAngleOfAttackWarning,
      DisplayedRollTrimPosition, DisplayedDirectionalTrimPos,
      AircraftAttitudes, ActualFlightPath, DisplayedInflightBrakePos,
      DisplayedConfigAndFailureStatus;
    description:
      { Receives a target flight path (generated by navigation) and generates
      control signals for the actuation systems which generate the forces and
      moments to control the aircraft attitudes to generate a flight path which
      matches the target flight path. }

    [** Requirement attributes: **}
    CMF: 1 General Control Requirements
    CMF: 2 Handling Qualities
    CMF: 3 Operational Flight Envelope
    CMF: 4 Manual and Automatic Trim Functions
    CMF: 5 Envelope protection
    CMF: 6 Autopilot Limiting and Actuation
    CMF: 7 Maneuver Control Lags
    CMF: 8 Requirements in Icing Conditions
    CMF: 9 Control System Stability Requirement
    CMF: 10 Residual Oscillations
    CMF: 11 Longitudinal Control Power Requirements
    CMF: 12 Longitudinal Trim Authority
    CMF: 13 Enhanced Longitudinal Control Maneuver Response
    CMF: 14 Roll Mode Time Constant
    CMF: 15 Pilot-Induced Oscillations
    CMF: 16 Stall Characteristics
    CMF: 17 Lateral Control Power Requirements
    CMF: 18 Roll Response Linearity
    CMF: 19 Roll Control Cross Coupling
    CMF: 20 Lateral Trim Authority
    CMF: 21 Enhanced Roll Maneuver Control
    CMF: 22 Dynamic Stability
    CMF: 23 Turn Coordination
    CMF: 24 Directional Control Power Requirements
    CMF: 25 Directional Trim Authority
    CMF: 26 Flutter Prevention Requirements

  end ControlMissionFlight;



 DSBP: 1a
  Means shall be provided to indicate to the flight crew the position of the
  speed brake system.

 DSBP: 1b
  Annunciation of failures or system operation which could result in an
  unsafe condition if the crew were not aware of the condition shall be
  provided (FAR 25..672a)

 DSBP: 1c
  Annunciation to the crew (in the form of an aural warning) shall be
  provided for speedbrake deployment or the following condition: take-off
  power and airplane on ground. (FAR 25.703a)

end;

  [** The following are atomic component operations of ControlMissionFlight.
      The remaining component operations are at the head of their respective
      modules. **}
  operation ControlThrust is
    inputs: { Note that there should probably be inputs here. };
    outputs: ThrustVectorActuatorConfiguration;
    description:
      { No description in ASCT };
  end ControlThrust;

  operation ControlHeadingOnGround is
    components: ;
    inputs: { Ibid. };
```

```
   outputs: NosewheelPosition;
   description:
     { No description in ASCT.  Note also that lack of inputs is suspicious };
 end ControlHeadingOnGround;

 operation ControlBrakingOnGround is
   components: ;
   inputs: { Ibid.};
   outputs: WhellBrakingPosition;
   description:
     { No description in ASCT.  Note also that lack of inputs is suspicious };
 end ControlBrakingOnGround;

 {** Organizationally, this operation would probably better be included in the
     Aircraft module.  It is here to maintain some lexical correspondence with
     ASCT. **}
 operation UpdateAircraftState is
   components: ;
   inputs: ThrustVectorActuatorConfiguration, PitchActuatorPosition,
       RollActuatorPosition, YawActuatorPosition, NosewheelPosition,
       DragActuatorPosition, WheelBrakePosition, LiftConfig, Aircraft.State
    outputs: AircraftAttitudes, ActualFlightPath, Aircraft.State;
   description: ;
     { Includes the airframe and the flight environment and outputs the
     aircraft flight state as a result of the flight state and the
     configuration of the flight control system.  Note: this appears to be a
     rather imprecise description; furthermore, inputs and outputs are not
     clearly specified.  See translation notes for further discussion. };
 end UpdateAircraftState;

 {** The following are atomic operations of Control Mission Flight, from pp.
     87-88 of ASCT.  Global non-atomic operations, such as Aircraft.State,
     ActualFlightPath, and TargetFlightPath are defined in appropriate major
     object modules.  Local objects that belong to functions defined in other
     modules, such as PitchActuatorPosition, are defined in the appropriate
     operation modules.  Use the browser to find their definitions. **}

 object ThrustVectorActuatorConfiguration is
   components: ;
   operations: ;
   description:
     { Configuration of the system which controls the magnitude and direction
     of the thrust vector.};
 end ThrustVectorActuatorConfiguration;
```

```
{** General Control Requirements (C.M.F.1), Pg. 18: Two modes of manual
    control shall be provided: core control and enhanced control.  **}

object ManualControlMode is
  components: CoreControl or EnhancedControl
  description:
    {The core control mode provides the minimum level of augmentation (e.g.,
    yaw damper, Mach trim, etc.) required by FAA certification at all failure
    levels not extremely improbable (probability < 1.0E-9).
end;

axiom

{*** Pg. 18: Transfer between core and
operation TransferControlMode
  inputs: AS: Aircraft.State
  outputs: AS': Aircraft.State
  postcond: if AS.ManualControlMode = CoreControl
            then AS'.ManualControlMode = EnhancedControl
  agent: Crew or AutoControlUnit

object CoreControl is "CoreControl";
object EnhancedControl is "EnhancedControl";

object HandlingQuality is
  components: Normal or Degraded;
end HandlingQuality;

object Normal is "Normal";
object Degraded is "Degraded";

end ControlMissionFlight;
```

```
{*****************************************************************************
 * Module ControlAerodynamicBraking from pp. 90-107
 *****************************************************************************}
module ControlAerodynamicBraking

  operation field CAB;

  operation ControlAerodynamicBraking is {** pg. 88, 105-107 **}
    components:
      GenerateManualBrakeCommand, GenerateAutoBrakeCommand,
      DisplaySpeedBrakePos, MoveDragActuator, ProvideCrewBrakingInterface
      GenerateDragActuatorCommand;
    inputs: TargetFlightPath, ExternalForcessOnActuator, ActualFlightPath
    outputs: DragActuatorPosition, DisplayedInflightBrakePos, DragActuatorDisplay
    description:
      { Controls drag and lift dumping to provide and aerodynamic braking
     capability. }

    CAB: {pg. 90
     Manual and automatic control of aerodynamic braking shall be available.
     Manual control shall be able to override the automatic control function.
     Aerodynamic speed brake control function shall be available
 for on-ground and in-flight operation}

    CAB: {1.0 Ground Speed Brake Control
     Ground speedbrake control shall provide ground deceleration capability
 consistent with operational field landing length requirements.}

    CAB: { 1.2.0a Inflight Speed Brake Control
     The inflight speed brake actuators shall be sized to give adequate
 inflight deflection at Vmo/Mmo for emergency descent.}

    CAB: { 1.2.0b Inflight Speed Brake Control
     Normal descent speed brake requirements shall not cause objectionable
 horizontal tail buffet of engine flow distortion (FAR 25.251b)}

    CAB: {1.2.0c CAB 2.0c Inflight Speed Brake Control
     Control forces to trim the pitching moment change shall be less
 than or equal to those required by FAR 25.143(b)}

    CAB: {2a  Aerodynamic Braking Function Availability Requirements }
     Each individual speed brake device shall provide fail-passive control for
     failure modes more probable than 10-7/flt hour}

    CAB: { 2b
```

```
   Loss of all speedbrake control shall be less than 10-7/flt hour}
end ControlAerodynamicBraking;

operation GenerateManualBrakeCommand is
  components: ;
  inputs: TargetFlightPath, ActualFlightPath;
  outputs: CrewBrakeForce;
  agent: Crew;
  description:
    { Generates the speedbrake command manually (i.e., by the crew).}
end GenerateManualBrakeCommand;

operation GenerateAutoBrakeCommand is
  components: ;
  inputs: TargetFlightPath, ActualFlightPath;
  outputs: AutoBrakeCommand;
  agent: FlightControlSystem;
  description:
    {Involves generation of the speedbrake command in an automated fashion};
end GenerateAutoBrakeCommand;

operation DisplaySpeedBrakePos is
  components: ;
  inputs: DragActuatorDisplay;
  outputs: DisplayedInflightBrakePos;
  description: ;
    { Indicates to the flight crew the position of the speedbrake system and
    annunciates unsafe speedbrake positions and unsafe failures. }
end DisplaySpeedBrakePos;

operation MoveDragActuator is
  components: ;
  inputs: DesiredDragActuatorPosition, ExternalForcesOnActuator;
  outputs: DragActuatorDisplay;
  description:
    {Moves the position of the system which provides the aerodynamic braking
    and lift dumping capability (spoiler/speedbrakes)};
end MoveDragActuator;

operation ProvideCrewBrakingInterface is
  components: ;
  inputs: CrewBrakeForce;
  outputs: ManualBrakeCommand;
  description:
    {Converts the force exerted by the crew into an aerodynamic braking
```

```
      command};
  end ProvideCrewBrakingInterface;

  operation GenerateDragActuatorCommand is
    components: ;
    inputs: ManualBrakeCommand, AutoBrakeCommand;
    outputs: DesiredDragActuatorPosition;
    description:
      { Generates a drag actuator command based on the manual and auto braking
      commands};
  end GenerateDragActuatorCommand;

end ControlAerodynamicBraking;
```

```
{****************************************************************************
 * Module ControlLiftConfiguration from pp. 92-119
 ****************************************************************************}
module ControlLiftConfiguration

  operation field CLC "label comment"

  operation ControlLiftConfig is
    components:
    inputs: TargetFlightPath, ActualFlightPath
    outputs: DisplayedConfigAndFailureStatus, LiftConfig
    description:
      { Configures the wing for different lift properties such that required
      lift and control is achieved at low speed (takeoff and landing) and low
      drag an be achieved at high speeds. }

   CLC: { 1
    The wing high lift design (both leading edge and trailing edge devices)
    shall be adjustable to provide a variable lift capability to ensure the
    achievement of low speeds performance requirements coupled with certifiable
    handling characteristics. Manual and automatic system operation shall be
    provided. High lift device position indication and failure status shall be
    available.}

   CLC: {2  p. 93. Lift configuration control function availability
    requirements. The high lift system shall provide the following functional
    availability ( function, probability of loss of function
    (LE and TE Control, 10-7)
    (LE Control, 10-6)
    (TE Control, 10-6)
    (Autoslat, 10-5)
    (Flap load relief, 10-5)
    (LE and TE Failure annunciation, 10-5)
    (LE Control and LE Failure annunciation, 10-9)
    (TE Control and TE Failure annunciation, 10-9)}

  end ControlLiftConfig;

  operation GenerateMaualConfigCmd is
    components: ;
    inputs: TargetFlightPath, ActualFlightPath;
    outputs: CrewHLConfigCmdForce;
    agent: Crew;
    description:
      { Involves the generation of the high lift configuration command in the
    manual fashion (i.e., by the crew). Note that name spelling (...Cmd) is
    not consistent with spellings of comparable operations (i.e.,
    ...Command).}
  end GenerateMaualConfigCmd;

  operation GenerateAutoConfigCommand is
    components: ;
    inputs: TargetFlightPath, ActualFlightPath;
    outputs: AutoConfigCommand;
    agent: MissionFlightSystem;
    description:
      { Generates the high lift configuration command in an automated fashion
      (i.e., by the computer system). }
  end GenerateAutoConfigCommand;

  operation DisplayConfigAndFailStatus is
    components: ;
    inputs: HighLiftConfigAndFailureStatus;
    outputs: DisplayedConfigAndFailureStatus;
    description:
      { Displays to the crew the position of the high lift devices and
      annunciates any height lift device failure conditions. };
  end DisplayConfigAndFailStatus;

  operation MoveLiftConfigActuator is
    components: ;
    inputs: HighLiftActuatorCommands;
    outputs: HighLiftDevicePositions, HighLiftConfigAndFailureStatus, LiftConfig
    description:
      { Involves the actuation of the high lift devices (i.e., the leading edge
      and trailing edge flaps).
  end MoveLiftConfigActuator;

  operation ProvideCrewConfigInterface is
    components: ;
    inputs: CrewHLConfigCmdForce;
    outputs: ManualConfigCmd;
    description:
      { Provides the interface which allows the crew to input commands to the
      high lift system.  See notes in analysis section about ad hoc user
      interface specification in the original ASCT.};
  end ProvideCrewConfigInterface;

  operation GenerateConfigActuatorCmd is
    components: ;
    inputs: AutoConfigCommand, HighLiftDevicePositions, ManualConfigCmd;
    outputs: HighLiftActuatorCommands;
    description:
      { Involves the actuation of the high lift devices (i.e., the leading edge
      and trailing edge flaps). };
  end GenerateConfigActuatorCmd;

end ControlLiftConfiguration;
```

```
{*****************************************************************************
 * Module ControlPitch from pp. 120-144
 *****************************************************************************}
module ControlPitch
  from FlyMission import ExternalForcesOnActuator, TargetFlightPath;

  operation field LAPC "label comment"
  operation field PLEP "label comment"
  operation field PSAW "label comment"

  operation ControlPitch is {pp. 87, 120}
    components: GenerateLongitudinalTrimCommand, DisplayLongitudinalTrimStatus,
      GeneratePitchActuatorCommand, MovePitchActuators,
      ProvideStallAngleOfAttackWarning, ProvideLongitudinalEnvelopeProtection,
      GenerateFlightPathCommand, LimitAutoPitchCommand;
    inputs: ActualFlightPath, ExternalForcesOnPitchActuator, TargetFlightPath
    outputs: StallAngleOfAttackWarning, DisplayedLongitudinalTrimPosition,
        PitchActuatorPosition
    description:
      { Performs all functions required to control the lateral axis by
      controlling the pitch angle. }
     LAPC: 1
     PLEP: 1
     PLEP: 2
     PSAW: 1
  end ControlPitch;

  operation <NAME> is

  operation GenerateLongitudinalTrimCommand is
    components:
    inputs: ;
    outputs: AutoLongitudinalTrimCommand, ManualLongitudinalTrimCommand;
    description:
      { Generates trim commands to offload steady state pitch commands from the
      elevator to the stabilizer. }

  end GenerateLongitudinalTrimCommand;

  operation DisplayLongitudinalTrimStatus is
    components:
    inputs: LongitudinalTrimPosition
    outputs: DisplayedLongitudinalTrimPosition
    description:
      { Displays the longitudinal trim status to the crew.  NOTE: inconsistency
    in this function name on pp. 120 and 121.) }
end DisplayLongitudinalTrimStatus;

operation GeneratePitchActuatorCommand is
  components:
  inputs: LimitedFlightPathCommand and ManualLongitudinalTrimCommand and
  AutoLongitudinalTrimCommand and ActualFlightPath
  outputs: DesiredPitchActuatorPosition and LongitudinalTrimPosition
  description:
    { Generates the pitch actuator (elevator and stabilizer) position command
    based on the flight path angle and longitudinal trim commands
end GeneratePitchActuatorCommand;

operation MovePitchActuators is
  components:
  inputs: DesiredPitchActuators, ExternalForcesOnActuator
  outputs: PitchActuatorPosition
  description:
    { Receives the desired pitch actuators positions and attempts to move the
    actuators to thoses positions. }
end MovePitchActuators;

operation ProvideStallAngleOfAttackWarning is
  components:
  inputs: ActualFlightPath
  outputs: StallAngleOfAttackWarning
    description:
    { Monitors the aircraft flight path state vector and attitudes and
    generates a warning for the crew when approaching the aircraft stall
    angle of attack.  NOTE naming inconsistency on pp. 120 and 121.}
end ProvideStallAngleOfAttackWarning;

operation ProvideLongitudinalEnvelopeProtection is
  components:
  inputs: ActualFlightPath, ManualFlightPathCommand, LimitedFlightPathCommand
  outputs: LimitedFlightPathCommand
  description:
    { Monitors the aircraft states and modifies the flight path angle command
    as necessary to satisfy the longitudinal envelope protection
    requirements. }
end ProvideLongitudinalEnvelopeProtection;

operation GenerateFlightPathCommand is
  components:
    GenerateFlightPathCommandManual, MakeManualVsAutoFlightModeDecision,
        EngageManOrAutoOperation, GenerateFlightPathCommandAuto
      inputs: TargetFlightPath, ActualFlightPath
      outputs: ManualFlightPathCommand, AutoFlightPathCommand
      description:
        { Compares the actual flight path angle to the desired flight path angle
        and generates the necessary flight path angle command. }
    end GenerateFlightPathCommand;

    operation LimitAutoPitchCommand is
      components:
      inputs: AutoFlightPathCommand
      outputs: LimitedFlightPathCommand
      description:
        { Limits the autopilot control authority and protects against failures
        (in particular hardover and oscillatory failures) in the autopilot. }
    end LimitAutoPitchCommand;


    object DisplayedLongitudinalTrimPosition is

      components:
      operations:
      description:
        { The longitudinal trim position displayed to the crew }
    end DisplayedLongitudinalTrimPosition;

    object AutoLongitudinalTrimCommand is

      components:
      operations:
      description:
{ The Longitudinal trim command generated automatically during enhanced
manual control and autoflight control }
    end AutoLongitudinalTrimCommand;

    object ManualLongitudinalTrimCommand is

      components:
      operations:
      description:
{ The longitudinal trim command generated by the crew for use during
normal and backup control }
    end ManualLongitudinalTrimCommand;

    object AutoFlightPathCommand is
    components:
    operations:
    description:
      { The flight path command generated automatically during enhanced manual
control and autoflight control }
  end AutoFlightPathCommand;

  object LongitudinalTrimPosition is

    components:
    operations:
    description:
{ Position of the longitudinal trim actuator }
  end LongitudinalTrimPosition;

  object ActualFlightPath is

    components:
    operations:
    description:
{ The sensed 4 dimensional flight path & attitudes of the aircraft as
well as other sensed values necessary to satisfy the control
requirements.}
  end ActualFlightPath;

  object ManualFlightPathCommand is

    components:
    operations:
    description:
{ The Flight path angle command generated manually (i.e., by the crew)}
  end ManualFlightPathCommand;

  object LimitedFlightPathCommand is

    components:
    operations:
    description:
{ The flight path command limited such that envelope protection is not
violated}
  end LimitedFlightPathCommand;

  object StallAngleOfAttackWarning is
```

```
    components:
    operations:
    description:
{ The audible and visual indication to the crew that the aircraft is
approaching the stall angle of attack}
  end StallAngleOfAttackWarning;

  object PitchActuatorPosition is

    components:
    operations:
    description:
{ The Position of the actuator(s) which provide(s) aircraft pitch
maneuver and trim control. }
  end PitchActuatorPosition;

  object TargetFlightPath = FlyMission.TargetFlightPath;

  object DesiredPitchActuatorPosition is
    components:
    operations:
    description:
      { The desired pitch actuator (elevator) position such that the limited
      flight path angle command is achieved }
  end DesiredPitchActuatorPosition;

  object ExternalForcesOnActuator = FlyMission.ExternalForcesOnActuator;

  { Pp. 127 - 128 }
  operation GenerateFlightPathCmdManual is
    components: ;
    inputs: ActualFlightPath, TargetFlightPath;
    outputs: ManualFlightPathCommand;
    description:
      { Involves the generation of a flight path command manually (i.e., by the
      crew) as a result of comparing the target and actual flight paths. } ;
  end GenerateFlightPathCmdManual;

  operation GenerateFlightPathCmdAuto is
    components: ;
    inputs: ;
    outputs: ;
    description:
      { Generates a flight path angle command automatically (i.e., by the a
      computer) as a result of the difference between the actual and target
```

```
        flight< paths. };
    end GenerateFlightPathCmdAuto;

    operation MakeManualVsAutoFlightModeDecision is
      components: ;
      inputs: ;
      outputs: ManualFlightMode;
      description: AutoFlightMode;
        { Decides whether to generate flight path commands manually or
        automatically. };
    end MakeManualVsAutoFlightModeDecision;

    operation EngageManOrAutoOperation is
      components: ;
      inputs: ManualFlightMode;
      outputs: AutoFlightMode;
      description:
        { Activates one of the flight path command generation processes depending
        on the mode engaged. };
    end EngageManOrAutoOperation;

end ControlPitch;
```

```
{*****************************************************************************
 * Module FlightControlSystemPitchFunctions from pp. 129-144
 *****************************************************************************}
module FlightControlSystemPitchFunctions;

  operation FlightControlSystemPitchContext is
    components: PerformPilotFunctions, PerformCopilotFunctions,
      FlightControlSystemPitchFunctions, PerformAutoFlightSystemFunctions;
    inputs:
      { Unclear - see pg. 129. };
    outputs: PitchActuatorPosition;
    description: { Unclear - see pp. 129-130. };
  end FlightControlSystemPitchContext;

  operation FlightControlSystemPitchFunctions is
    components: ProvidePilotPitchInterface, ProvideCopilotPitchInterface,
      DisplayLongitudinalTrimStatus, ResolvePitchControlContention,
      GeneratePitchActuatorCommand, MovePitchActuators,
      ProvideStallAngleOfAttackWarning,
      DisplayLongitudinalEnvelopeProtectStatus,
      ProvideLongitudinalEnvelopeProtection, LimitAutoPitchCommands,
    inputs: PilotLongitudinalTrimForce, PilotFlightPathCommandForce,
      CopilotLongitudinalTrimForce, CopilotFlightPathCmdForce,
      AutoLongitudinalTrimCommand, AutoFlightPathCommand
      { PLUS MAYBE THE FOLLOWING DUE TO AMBIGUITY ON PP. 129 VS 133: }
      , ActualFlightPath, ExternalForcesOnActuator;
    outputs: PilotFlightPathCmdFeelForce, CopilotFlightPathCmdFeelForce,
      PitchActuatorPosition
    description:
      { Note that the following description is taken from ASCT page 130, but
      it is not a fully accurate description of this operation as it is defined
      in WSRSL.  See the remarks on page ??? of the report.

      Contains all the flight control functions assigned to the FCS.  As a
      result of this assignment several new processes are created.  some of
      these are interface functions and others are as a result of how functions
      were allocated to the AEs.  (I.e., Envelope Protection was assigned to the
      FCS with a probability of failure < 10E-6.  However this function
      requires <10E-9.  Therefore the pilot and copilot must perform envelope
      protection when not being performed by the FCS.  Thus a pilot indication
      function of the status of envelope protection is generated.)  Pilot and
      copilot can command roll reate, thus there is a function requirement to
      resolve control contention.}
end FlightControlSystemPitchFunctions;
```

```
  object CopilotFlightPathCommandFeelForce is
    components:
    description:
{ A resistance force exerted by the controller which is a feedback to the
copilot indicative of the flight path angle. }
  end CopilotFlightPathCommandFeelForce;

  object CopilotFlightPathCommandForce is
    components:
    description:
{ The physical force generated by the copilot to control the aircraft
flight path angle. It is in the form of a force exerted by the pilot's
hand.}
  end CopilotFlightPathCommandForce;

  object CopilotLongitudinalTrimForce is
    components:
    description:
      { The physical force exerted by the copilot's hand to generate the
      desired longitudinal trim command. };
  end CopilotLongitudinalTrimForce;

  object PilotFlightPathCommandForce is
    components:
    description:
{ The physical signal created by the pilot to control the aircraft flight
path. It is in the form exerted by the pilot.}
  end PilotFlightPathCommandForce;

  object PilotFlightPathFeelForce is
    components:
    description:
      { A resistance force exerted by the controller which is a feedback to the
      pilot indicative of the flight path command.}
  end PilotFlightPathFeelForce ;

  object PilotLongitudinalTrimForce is
    components:
    description:
      {This flow is the physical force exerted by the pilot's hand to generate
      the desired longitudinal trim command.}
  end PilotLongitudinalTrimForce;

  object PitchActuatorPosition is
    components:
```

```
  description:
    { Position of the actuator(s) which provide aircraft pitch maneuver and
    trim control.}
end PitchActuatorPosition;

{Pp. 133 - 135}
operation ProvidePilotPitchInterface is
  components: ;
  inputs: PilotLongitudinalTrimForce, PilotFlightPathCmdForce;
  outputs: PilotFlightPathCmdFeelForce, PilotLongitudinalTrimCommand,
    PilotFlightPathCommand;
  description:
    { Converts the signal received from the pilot in the form of a force
    exerted by the pilot into a flight path angle command signal to be used
    by the FCS.  It also provides the pilot with a feedback feel force
    indicative of the command. };
end ProvidePilotPitchInterface

operation ProvideCopilotPitchInterface is
  components: ConvertForceToDisplacement, GenerateLongitudinalFeelForce,
    TranslateFlightPathDisplacementToCommand,
    TranslateTrimForceToTrimCommand;
  inputs: CopilotFlightPathCmdForce, CopilotLongitudinalTrimForce;
  outputs: CopilotFlightPathCmdFeelForce, CopilotFlightPathCommand,
    CopilotLong<itudinalTrimCommand ;
   description:
    { Provides the same capability for the copilot as the
    ProvidePilotPitchInterface does for the pilot. };
end ProvideCopilotPitchInterface

operation DisplayLongitudinalTrimStatus is
  components: ;
  inputs: ;
  outputs: LongitudinalTrimPosition;
   description:
    { Displays the longitudinal trim status to the crew. };
end DisplayLongitudinalTrimStatus

operation ResolvePitchControlContention is
  components: ;
  inputs: CopilotFlightPathCommand, CopilotLongitudinalTrimCommand,
    PilotFlightPathCommand, PilotLongitudinalTrimCommand;
  outputs: ManualFlightPathCommand;
   description:
    { Generated by the assignment of the GenerateFlightPathCommandManual to
    both the pilot and copilot.  NOTE: this description is unclear. };
end ResolvePitchControlContention

operation GeneratePitchActuatorCommand is
  components: ;
  inputs: ManualFlightPathCommand, ActualFlightPath,
    AutoLongitudinalTrimCommand;
  outputs: LongitudinalTrimPosition, DesiredPitchActuatorPosition;
   description:
    { Generates the pitch actuator (elevator and stabilizer) position
    commands based on the flight path angle and longitudinal trim commands.
    };
end GeneratePitchActuatorCommand

operation MovePitchActuators is
  components: ;
  inputs: DesiredPitchActuatorPosition, ExternalForcesOnActuator;
  outputs: PitchActuatorPosition;
   description:
    { Receives the desired pitch actuators positions and attempts to move the
    actuators to those positions. };
end MovePitchActuators

operation ProvideStallAngleOfAttackWarning is
  components: ;
  inputs: ActualFlightPath;
  outputs: StallAngleOfAttackWarning;
   description:
    { Monitors the aircraft flight path state vector and attitudes and
    generates a warning for the crew when approaching the aircraft stall
    angle of atack. };
end ProvideStallAngleOfAttackWarning

{NOTE: Inconsistent Names Pp. 133, 134}
operation DisplayLongitudinalEnvelopeProtectStatus is
  components: ;
  inputs: LongitudinalEnvelopeProtectStatus;
  outputs: DisplayedLongitudinalEnvelopeProtectStatus;
   description:
    { Results from the allocation of ProvideLongitudinalEnvelopeProtection to
    the FCS with a probability of loss of function of <10E-6.  Pitch envelope
    protection has a req for probability of loss of function <10E-9, and
    thus the crew has responsibility for pitch envelope protection when not
    performed by the FCS.  Thus the crew must be aware of envelope protect
    status, hence the functional requirement to
    DisplayLongitudinalEnvelopeProtectStatus  };
end DisplayLongitudinalEnvelopeProtectStatus

operation ProvideLongitudinalEnvelopeProtection is
  components: ;
  inputs: ActualFlightPath, LimitedFlightPathCommand,
    ManualFlightPathCommand;
  outputs: LimitedFlightPathCommand, LongitudinalEnvelopeProtectStatus;
   description:
    { Monitors the aircraft states and modifies the flight path angle command
    as necessary to satisfy the longitudinal envelope protection requirements.
    };
end ProvideLongitudinalEnvelopeProtection

operation LimitAutoPitchCommands is
  components: ;
  inputs: AutoFlightPathCommand;
  outputs: LimitedFlightPathCommand;
   description:
    { Limits the autopilot control authority and protects against failures
    (in particular hardover and oscillatory failures in the autopilot. };
end LimitAutoPitchCommands

{*** Pilot Pitch Interface, pp. 137-138 ***}
operation ConvertForcesToDisplacement is
  components: ;
  inputs: FlightPathCommandForce, FlightPathCommandFeelForce;
  outputs: FlightPathCommandDisplacement;
  description:
    { Receives the pilot force and feedback feel force and generates a
    displacement.  Note name inconsistency on pp. 137, 138. }
end ConvertForcesToDisplacement;

operation GenerateLongitudinalFeelForce is
  components: ;
  inputs: FlightPathAngleCommand;
  outputs: FlightPathCommandFeelForce;
  description:
    { Generates a force to feedback to the pilot which is indicative of the
    pitch maneuver and trim commands.  Note name inconsistency pp. 137,138. }
end GenerateLongitudinalFeelForce;

operation TranslateFlightPathDisplacementToCommand is
  components: ;
  inputs: FlightPathCommandDisplacement;
```

```
    outputs: FlightPathAngleCommand;
    description:
      { Translates the physical displacement of the pitch controller into a
      flight path command.  Note name inconsistency pp. 137,138. }
  end TranslateFlightPathDisplacementToCommand;

  operation TranslateTrimForceToTrimCommand is
    components: ;
    inputs: LongitudinalTrimForce;
    outputs: LongitudinalTrimCommand;
    description:
      { Converts the physical displacement generated by the physical force
      exerted by the pilot into a trim command for use by the FCS. Note name
      inconsistency pp. 137,138. }
  end TranslateTrimForceToTrimCommand;

end FlightControlSystemPitchFunctions;
```

```
{****************************************************************************
 * Module ControlRoll from pp. 145-168
 ****************************************************************************}
module ControlRoll

  operation ControlRoll is {pp. 87, 145}
    components: GenerateRollTrimCommand, DisplayRollTrimPosition,
      GenerateRollActuatorCommand, MoveRollActuator,
      ProvideRollEnvelopeProtection, GenerateRollRateCommand,
      LimitAutoRollCommands;
    inputs: TargetFlightPath, ActualFlightPath, ExternalForcesOnActuator;
    outputs: DisplayedLongitudinalTrimPosition, TargetFlightPath,
        RollActuatorPosition, ExternalForcesOnYawActuator;
    description:
     { Performs all functions required to control the lateral axis by
     controlling the roll angle. };
  end ControlRoll;

  operation GenerateRollTrimCommand is
    components: ;
    inputs: {Note that no inputs is suspicious here};
    outputs: AutoRollTrimCommand, ManualRollTrimCommand;
    description:
      { Generates roll trim commands to offset asymmetries such as engine
      out, engine loss and lateral winds. }
  end GenerateRollTrimCommand;

  operation DisplayRollTrimPosition is
    components: ;
    inputs: RollTrimPosition;
    outputs: DisplayedRollTrimPosition;
    description:
      { Displays roll trim position to the crew. };
  end DisplayRollTrimPosition;

  operation GenerateRollActuatorCommand is
    components: ;
    inputs: ManualRollTrimCommand, AutoRollTrimCommand, ActualFlightPath,
      LimitedRollRateCommand;
    outputs: RollTrimPosition, DesiredRollActuatorPosition;
    description:
      { Generates the roll actuator (aileron / spoiler) position commands based
      on roll rate and trim commands. };
  end GenerateRollActuatorCommand;
```

```
operation MoveRollActuator is
  components: ;
  inputs: ;DesiredRollActuatorPosition, ExternalForcesOnActuator;
  outputs: RollActuatorPosition;
  description:
    { Receives the desired roll actuator position and attempts to move the
    roll actuator to that position. };
end MoveRollActuator;

operation ProvideRollEnvelopeProtection is
  components: ;
  inputs: ManualRollRateCommand, LimitedAutoRollCommand, RollAngle;
  outputs: LimitedRollRateCommand;
  description:
    { Monitors actual roll angle and commanded roll rate and modifies the
    roll rate command as necessary to prevent the roll angle from exceeding
    certain limits. };
end ProvideRollEnvelopeProtection;

operation GenerateRollRateCommand is
  components: GenerateRollRateCommandManual, EngageManOrAutoOperation,
    GenerateRollRateCommandAuto, MakeManualVsAutoFlightModeDecision;
  inputs: TargetFlightPath, ActualFlightPath;
  outputs: AutoRollRateCommand, ManualRollRateCommand;
  description:
    { Compares the target flight path and actual flight path and generates
    necessary roll rate command to drive the actual to the target. };
end GenerateRollRateCommand;

operation LimitAutoRollCommands is
  components: ;
  inputs: AutoRollRateCommand;
  outputs: LimitedAutoRollCommand;
  description:
    { Limits the autopilot control authority and protects against failures
    (in particular hardover or oscillatory failures) in the autopilot. };
end LimitAutoRollCommands;

{ Pp. 151-152 }
operation GenerateRollRateCommandManual is
  components: ;
  inputs: ActualFlightPath, TargetFlightPath, ManualModeEngaged;
  outputs: ManualRollRateCommand;
  description:
    { Involves the generation of ta roll rate command manually (i.e., by the
    crew) as a result of comparing the target and actual flight paths. };
  end GenerateRollRateCommandManual;

  operation EngageManOrAutoOperation is
    components: ;
    inputs: ManualFlightMode, AutoFlightMode;
    outputs: ManualModeEngaged, AutoModeEngaged;
    description:
      { Activates one of the roll rate generation processes depending on the
      mode engaged. }
  end EngageManOrAutoOperation;

  operation GenerateRollRateCommandAuto is
    components: ;
    inputs: AutoModeEngaged, TargetFlightPath, ActualFlightPath;
    outputs: AutoRollRateCommand;
    description:
      { Involves the generation of a roll rate command automatically (i.e., by
      the computer) as a result of the difference between the actual and target
      flight path. };
  end GenerateRollRateCommandAuto;

  operation MakeManualVsAutoFlightModeDecision is
    components: ;
    inputs: { Note no inputs - seem reasonable here. };
    outputs: ManualFlightMode, AutoFlightMode;
    description:
      { Decides whether to generate flight path commands manually or
      automatically.  Note - not clear if this should be the same as operation
      of the same in ControlPitch module. };
  end MakeManualVsAutoFlightModeDecision;

end ControlRoll;
```

```
{*****************************************************************************
 * Module FlightControlSystemRollFunctions from pp. 153-168
 *****************************************************************************}
module FlightControlSystemRollFunctions

  operation FlightControlSystemRollContext is
    components: PerformPilotFunctions, PerformCopilotFunctions,
      FlightControlSystemRollFunctions, PerformAutoFlightSystemFunctions;
    inputs: { Unclear - see pg. 153. };
    outputs: RollActuatorPosition;
    description: { Unclear - see pp. 153-154.  Also cf.
      FlightControlSystemPitchContext in module
      FlightControlSystemRollFunctions above. };
  end FlightControlSystemRollContext;

  operation FlightControlSystemRollFunctions is
    components: ProvidePilotRollInterface, ProvideCopilotRollInterface,
      DisplayRollTrimStatus, ResolveRollControlContention,
      GenerateRollActuatorcommand, MoveRollActuator,
      DisplayRollEnvelopeProtectStatus, ProvideRollEnvelopeProtection,
      LimitAutoRollCommands;
    inputs: PilotRollTrimForce, PilotRollRateForce, CopilotRollRateForce,
      CopilotRollTrimForce, AutoRollTrimCmd, AutoRollRateCmd;
      { plus maybe the following due to ambiguity on pp. 153 versus 157: }
      , ActualFlightPath, ExternalForcesOnActuator, RollAngle {Note that the
      RollAngle input here is seemingly inconsistent with the ActualFlightPath
      input in the comparable position in the FlightContorlSystemPitchFunctions
      on pg. 133.};
    outputs: PilotRRCmdFeelForce, CopilotRRCmdFeelForce,
      DisplayedRollTrimPosition, RollActuatorPosition,
      DisplayedRollEnvelopeProtectStatus {Note that as with inputs, these are
      inconsistent on pp. 153 versus 157.};
    description:
      { Note that the following description is taken from ASCT page 154, but
      it is not a fully accurate description of this operation as it is defined
      in WSRSL.  See the remarks on page ??? of the report.  Cf. also
      description of operation FlightControlSystemPitchFunctions above.

      Contains all the flight control functions assigned to the FCS.  As a
      result of this assignment several new processes are created.  Some of
      these are interface functions and others are as a result of how
      functions were allocated to the AEs.  (I.e., Envelope protection was
      assigned to the FCS with a probability of failure <10E-6.  However this
      function requires <10E-09.  Therefore the pilot and copilot must perform
      envelope protection when not being performed by the FCS.  Thus a pilot
    indication function of the status of envelope protection is generated.)
    Pilot and copilot can command roll rate, thus there is a functional
    requirement to resolve control contention. };
end FlightControlSystemRollFunctions;

operation ProvidePilotRollInterface is
  components: ConvertForcesToDisplacement, GenerateRollFeelForce,
    TranslateRRDisplToRRCommand, TranslateTrimForceToTrimCommand;
  inputs: PilotRollTrimForce, PilotRollRateForce;
  outputs: PilotRRCmdFeelForce, PilotRollTrimCommand, PilotRollRateCommand;
  description:
    { Converts the signal received from the pilot in the form of a force
    exerted by the pilots hand into a roll rate signal to be used by the
    FCS.  It also provides the pilot with a feedback feel force proportional
    to the commanded roll rate. };
end ProvidePilotRollInterface;

operation ProvideCopilotRollInterface is
  components: ;
  inputs: CopilotRollRateForce, CopilotRollTrimForce;
  outputs: CopilotRRCmdFeelForce, CopilotRollRateCommand,
    CopilotRollTrimCommand;
  description:
    { Provides the same function for the copilot as the
    ProvidePilotRollInterface does for the pilot. };
end ProvideCopilotRollInterface;

operation DisplayRollTrimStatus is
  components: ;
  inputs: RollTrimPosition;
  outputs: DisplayedRollTrimPosition;
  description:
    { Displays roll trim position to the crew.  Note naming inconsistency on
    pp. 157 vs. 158. };
end DisplayRollTrimStatus;

operation ResolveRollControlContention is
  components: ;
  inputs: PilotRollRateCommand, PilotRollTrimCommand, CopilotRollRateCommand,
    CopilotRollTrimCommand;
  outputs: ManualRollRateCommand, ManualRollTrimCmd;
  description:
    { Generated by the assignment of the Generate Roll Rate Cmd Manual to
    both the pilot and copilot. }
end ResolveRollControlContention;
```

```
operation GenerateRollActuatorCommand is
  components: ;
  inputs: LimitedRollRateCommand, ManualRollTrimCmd, ActualFlightPath,
    AutoRollTrimCmd;
  outputs: RollTrimPosition, DesiredRollActuatorPos;
  description:
    { Generates the roll actuator (aileron / spoiler) position commands based
    on roll rate and trim commands. };
end GenerateRollActuatorcommand;

operation MoveRollActuator is
  components: ;
  inputs: DesiredRollActuatorPos, ExternalForcesOnActuator;
  outputs: RollActuatorPosition;
  description:
    { Receives the desired roll actuator position and attempts to move the
    roll actuator to that position. };
end MoveRollActuator;

operation DisplayRollEnvelopeProtectStatus is
  components: ;
  inputs: RollEnvelopeProtectStatus;
  outputs: DisplayedRollEnvelopeProtectStatus;
  description:
    { Results from the allocation of Provide Roll Envelope Protection to the
    FCS with a probability of loss of function of <10E-6.  Provide
    RollEnvelopeProtection has a probability of loss of function of < 10E-
9

    and thus the crew has responsibility for roll envelope protection when
    not performed by the FCS.  Thus the crew must be aware of envelope
    protect status, hence the function requirement to Display Roll Envelope
    Protect Status. };
end DisplayRollEnvelopeProtectStatus;

operation ProvideRollEnvelopeProtection is
  components: ;
  inputs: RollAngle {Note:Why not ActualFlightPath as in
    operation ProvideLongitudinalEnvelopeProtection on pg. 133},
    LimitedAutoRollcommand, ManualRollRateCommand;
  outputs: LimitedRollRateCommand, RollEnvelopeProtectStatus;
  description:
    { Monitors actual roll angle and commanded roll rate and modifies the
    roll rate command as necessary to prevent the roll angle from exceeding
    certain limits. }
```

```
end ProvideRollEnvelopeProtection;

operation LimitAutoRollCommands is
  components: ;
  inputs: AutoRollRateCommand;
  outputs: LimitedAutoRollCommand;
  description:
    { Limits the autopilot control authority and protects against failures
    (in particular hardover or oscillatory failures) in the autopilot. };
end LimitAutoRollCommands;

{*** Pilot Roll Interface, pp. 161-162 ***}
operation ConvertForcesToDisplacement is
  components: ;
  inputs: RollRateForce, RRCmdFeelForce;
  outputs: RollRateCmdDispl;
  description:
    { Receives the pilot force and feedback feel force and generates a
    displacement.};
end ConvertForcesToDisplacement;

operation GenerateRollFeelForce is
  components: ;
  inputs: RollRateCommand;
  outputs: RRCmdFeelForce, RRCmdFeelForce;
  description:
    { Generates a force to feedback to the pilot which is an indication of
    the commanded roll rate. };
end GenerateRollFeelForce;

operation TranslateRRDisplToRRCommand is
  components: ;
  inputs: RollRateCmdDispl;
  outputs: RollRateCommand, RollRateCommand;
  description:
    { Translates the sidestick controller displacement to a roll rate command.
    };
end TranslateRRDisplToRRCommand;

operation TranslateTrimForceToTrimCommand is
  components: ;
  inputs: RollTrimForce;
  outputs: RollTrimCommand;
  description:
    { Converts the physical displacement generated by the physical force
```

```
      exerted by the pilot into a trim command for use by the FCS. };
  end TranslateTrimForceToTrimCommand;

end FlightControlSystemRollFunctions;
```

```
{*****************************************************************************
 * Module ControlYaw from pp. 169 - 193
 *****************************************************************************}
module ControlYaw
  operation ControlYaw is
    components: GenerateDirectionalTrimCommand, DisplayDirectionalTrimPosition,
      GenerateYawActuatorCommand, EngineOutControlAugmentation,
      MoveYawActuator, ProvideYawEnvelopeProtection, GeneateSideslipCommand,
      LimitAutoSideslipCommands;
    inputs: TargetFlightPath, ActualFlightPath, EngineThrust,
      ExternalForcesOnYawActuator, SideslipAngle { Inconsistent pp. 87 versus
      169 };
    outputs: DisplayedDirectionalTrimPos, YawActuatorPosition;
    description:
      { Controls the aircraft directional axis. };
  end ControlYaw;

  operation GenerateDirectionalTrimCommand is
    components: ;
    inputs: { None - suspicious. };
    outputs: ManualDirectionalTrimCmd, AutoDirectionalTrimCmd;
    description:
      { Generates directional trim commands to offset asymmetries such as
      engine out and lateral winds.  Note: inconsistent names pp. 1.};
  end GenerateDirectionalTrimCommand;

  operation DisplayDirectionalTrimPosition is
    components: ;
    inputs: DirectionalTrimPosition;
    outputs: DisplayedDirectionalTrimPos;
    description:
      { Displays the position for the directional trim actuator to the crew. };
  end DisplayDirectionalTrimPosition;

  operation GenerateYawActuatorCommand is
    components: ;
    inputs: LimitedSideslipCommand, ManualDirectionalTrimCmd,
      AutoDirectionalTrimCmd, ActualFlightPath, ECAYawCommand;
    outputs: DirectionalTrimPosition, DesiredYawActuatorPosition;
    description:
      { Generates the sideslip actuator (rudder) position command based on the
      limited sideslip command, directional trim command and the engine out
      control augmentation command. };
  end GenerateYawActuatorCommand;
```

```
operation EngineOutControlAugmentation is
  components: ;
  inputs: EnginesThrust;
  outputs: ECAYawCommand;
  description:
    { Monitors the engine thrust and generates a yaw command to assist the
    pilot in compensation for an engine out situation.  In particular it
    helps relieve pilot workload in takeoff and go around which are high
    pilot workload situations. };
end EngineOutControlAugmentation;

operation MoveYawActuator is
  components: ;
  inputs: DesiredYawActuatorPosition, ExternalForcesOnYawActuator;
  outputs: YawActuatorPosition;
  description:
    { Receives the desired yaw actuator position and attempts to move the
    yaw actuator to that position. };
end MoveYawActuator;

operation ProvideYawEnvelopeProtection is
  components: ;
  inputs: SideslipAngle, ManualSideslipComand, LimitedAutoSideslipCommand;
  outputs: LimitedSideslipCommand;
  description:
    { Monitors the commanded sideslip and the actual sideslip and modifies
    the sideslip command to prevent the sideslip angle from exceeding unsafe
    limits. };
end ProvideYawEnvelopeProtection;

operation GeneateSideslipCommand is
  components: ;
  inputs: TargetFlightPath, ActualFlightPath;
  outputs: AutoSideslipCommand, ManualSideslipCommand;
  description:
    { Involves the generation of sideslip commands to allow for decrab for
    landings, performing coordinated turns and offsetting certain
    asymmetries. };
end GeneateSideslipCommand;

operation LimitAutoSideslipCommands is
  components: ;
  inputs: AutoSideslipCommand;
  outputs: LimitedAutoSideslipCommand;
```

```
    description:
      { Limits the autopilot control autority and protects against failures
      (in particular hardover or oscillatory failures) in the autopilot. };
  end LimitAutoSideslipCommands;

  operation GenerateSideslipCmdManual is
    components: ;
    inputs: ActualFlightPath, TargetntFlightPathNBManualSideslipCommand,
      ManualModeEngaged;
    outputs: ManualSideslipCommand;
    description:
      { Involves the generation of a sideslip command manually (i.e., by the
      crew) as a result of comparing the actual and desired flight path
      (including attitudes). };
  end GenerateSideslipCmdManual;

  { NOTE: Next to ops are generic and should, accordingly, appear in another
      module.  Cf. GenerateRollRateCommand (pg. 151) and
      GenerateFlightPathCommand (pg. 127
  operation MakeManualVsAutoFlightModeDecision is
    components: ;
    inputs: none;
    outputs: ManualFlightMode, AutoFlightMode;
    description: ;
  end MakeManualVsAutoFlightModeDecision;

  operation EngageManOrAutoOperation is
    components: ;
    inputs: ManualFlightMode, AutoFlightMode;
    outputs: ManualModeEngaged, AutoModeEngaged;
    description:
      { Involves the generation of a sideslip command automatically (i.e., by a
      computer. };
  end EngageManOrAutoOperation;

  operation GenerateSideslipCmdAuto is
    components: ;
    inputs: TargetFlightPath, ActualFlightPath, AutoModeEngaged;
    outputs: AutoSideslipCommand;
    description:
      { };
  end GenerateSideslipCmdAuto;

end ControlYaw;
```

```
{*****************************************************************************
 * Module FlightControlSystemYawFunctions from pp. 179-193
 *****************************************************************************}
module FlightControlSystemYawFunctions

  operation FlightControlSystemYawContext is
    components: PerformPilotFunctions, PerformCopilotFunctions,
      FlightControlSystemYawFunctions, PerformAutoFlightSystemFunctions;
    inputs: { Unclear - see pg. 179. } ;
    outputs: YawActuatorPosition;
    description: ;
  end FlightControlSystemYawContext;

  operation FlightControlSystemYawFunctions is
    components: ProvidePilotYawInterface, ProvideCopilotYawInterface,
      DisplayDirectionalTrimPosition, ResolveYawControlContentions,
      GenerateYawActuatorCommand, EngineOutControlAugmentation,
      MoveYawActuator, DisplayEnvelopeProtectStatus,
      ProvideYawEnvelopeProtection, LimitAutoSideslipCommands;
    inputs: PilotDirectionalTrimForce, PilotSideslipForce,
      CopilotDirectionalTrimForce, CopilotSideslipForce, AutoDirectionalTrimCmd,
      AutoSideslipCommand;
      { plus maybe the following due to ambiguity on pp. 179 versus 183: }
      , ActualFlightPath, EngineThrust, ExternalForcesOnActuator,
      SideslipAngle;
    outputs: ;
    description: ;
  end FlightControlSystemYawFunctions;

  operation ProvidePilotYawInterface is
    components: ConvertForceToDisplacement, GenerateSideslipFeelForce,
      TranslateSideslipDisplCmd, TranslateDirecTrimForceToCommand;
    inputs: PilotSideslipForce, PilotDirectionalTrimForce;
    outputs: PilotSideslipCmdFeelForce, PilotDirectionalTrimCmd,
      PilotSideslipCommand;
    description:
      { Converts the signal received from the pilot in the form of a force
      exerted by the pilot's hand into a sideslip signal to be used by the FCS.
      It also provides the pilot with a feedback force proportional to the
      command sideslip angle.};
  end ProvidePilotYawInterface;

  operation ProvideCopilotYawInterface is
    components: ;
    inputs: CopilotSideslipForce, CopilotDirectionalTrimForce,
    CopilotSideslipCommand, CopilotDirectionalTrimCmd;
  outputs: CopilotSideslipCmdFeelForce;
  description:
    { Provides the same function for the copilot as the
    ProvidePilotYawInterface does for the pilot. };
end ProvideCopilotYawInterface;

operation DisplayDirectionalTrimPosition is
  components: ;
  inputs: DirectionalTrimPosition;
  outputs: DisplayedDirectionalTrimPos;
  description:
    { Displays the position of the directional trim actuator to the crew.};
end DisplayDirectionalTrimPosition;

operation ResolveYawControlContentions is
  components: ;
  inputs: PilotSideslipCommand, PilotDirectionalTrimCmd,
    CopilotSideslipCommand, CopilotDirectionalTrimCmd;
  outputs: ManualSideslipCommand, ManualDirectionalTrimCmd;
  description:
    { Generated as a result of the assignment of the
    GenerateSideslipCmdManual to both the pilot and copilot. };
end ResolveYawControlContentions;

operation GenerateYawActuatorCommand is
  components: ;
  inputs: LimitedSideslipCommand, ManualDirectionalTrimCmd, ActualFlightPath,
    AutoDirectionalTrimCmd, ECAYawCommand;
  outputs: DirectionalTrimPosition, DesiredYawActuatorPosition;
 description:
    { Generates the sideslip actuator (rudder) position command based on the
    limited sideslip command, directional trim command and the engine out
    control augmentation command. };
end GenerateYawActuatorCommand;

operation EngineOutControlAugmentation is
  components: ;
  inputs: EnginesThrust;
  outputs: ECAYawCommand;
  description:
    { Monitors the engine thrust and generates a yaw command to assist the
    pilot in compensating for an engine out situation.  In particular it
    helps relieve pilot workload in takeoff and go around which are high
    pilot workload situations. };
```

```
end EngineOutControlAugmentation;

operation MoveYawActuator is
  components: ;
  inputs: DesiredYawActuatorPosition, ExternalForcesOnActuator;
  outputs: YawActuatorPosition;
  description:
    { Receives the desired yaw actuator position and attempts to move the yaw
    actuator to that position. };
end MoveYawActuator;

operation DisplayEnvelopeProtectStatus is
  components: ;
  inputs: YawEnvelopeProtectStatus;
  outputs: DisplayedYawEnvelopeProtectStatus;
  description:
    { Results from the allocation of ProvideYawEnvelopeProtection to the FCS
    with a probability of loss of function < 10E-6.  YawEnvelopeProtection
    has a proability of loss of function < 109E-9 and thus the crew has
    responsibility for yaw envelope protection when not performed by the FCS,
    hence the crew must be aware of the envelope protection status which
    leads to this functional requirement. }
end DisplayYawEnvelopeProtectStatus;

operation ProvideYawEnvelopeProtection is
  components: ;
  inputs: SideslipAngle, LimitedAutoSideslipCommand, ManualSideslipCommand;
  outputs: LimitedSideslipCommand, YawEnvelopeProtectStatus;
  description:
    { Monitors the commanded sideslip and the actual sideslip and modifies
    the sideslip command to prevent the sideslip angle from exceeding unsafe
    limits. };
end ProvideYawEnvelopeProtection;

operation LimitAutoSideslipCommands is
  components: ;
  inputs: AutoSideslipCommand;
  outputs: LimitedAutoSideslipCommand;
  description:
    { Limits the autopilot control authority and protects against failures
    (in particular hardover or oscillatory failures) in the autopilot. };
end LimitAutoSideslipCommands;

{*** Pilot Yaw Interface, pp. 187-188 ***}
```

```
  operation ConvertForceToDisplacement is
    components: ;
    inputs: SideslipForce, SideslipFeelForce;
    outputs: SideslipCommandDispl;
    description:
      { Receives the pilot force and feedback feel force and generates a
      displacement. };
  end ConvertForceToDisplacement;

  operation GenerateSideslipFeelForce is
    components: ;
    inputs: SideslipCommand;
    outputs: SideslipFeelForce;
    description:
      { Generates a force to feedback to the pilot which is an indication of the
      commanded sideslip angle. };
  end GenerateSideslipFeelForce;

  operation TranslateSideslipDisplCmd is
    components: ;
    inputs: SideslipCommandDispl;
    outputs: SideslipCommand;
    description:
      { Translates the displacement (rudder pedal) to a sideslip command. };
  end TranslateSideslipDisplCmd;

  operation TranslateDirecTrimForceToCommand is
    components: ;
    inputs: DirectionalTrimForce;
    outputs: DirectionalTrimCommand;
    description:
      { Converts the physical displacement generated by the physical force
      exerted by the pilot into a trim command for use by the FCS. };
  end TranslateDirecTrimForceToCommand;

end FlightControlSystemYawFunctions;
```

```
{*****************************************************************************
 * Module ControlAerodynamicBraking from pp. 198-219, and AE diagrams
 * pp. 218 - 219
*****************************************************************************}
module FlightControlSystem

  object FlightControlSystem is
    components: FlightControlComputer, SpeedBrakeController,
      HightLiftController, Displays, HightLIftSystem, RudderSystem,
      SpoilerSystem, AileronSystem, ElevatorStabilizerSystem,
      SidestickControllers, RudderPedals;
    description:
      { The primary agent, along with Crew members, to execute flight control
      operations };
  end FlightControlSystem;

  operation PerformAutoFlightSystemFunctions is
    components: ;
    inputs: ;
    outputs: ;
    description:
      { The AEAuto-FlightSystem ''Architectural Element". };
  end PerformAutoFlightSystemFunctions;


  object class Computer is
    components: ;
    operations: ;
    description: ;
  end Computer;

  object class Sensor is
    components: ;
    operations: ;
    description: ;
  end Sensor;

  object class SurfaceActuator is
    components: ;
    operations: ;
    description: ;
  end SurfaceActuator;

  object class Command is
    components: HowActuated, AffectedAircraftComponents, ... ;
    description:
      { The high-level class of control commands that are generated by either
      the crew or flight control system. }
  end Command;


  {**** Control System Signal Transmission (F.C.S.1), Pg. 199 ****}

  object Communicant is
    components: Computer | Sensor | SurfaceActuator;
    operations: ;
    description: ;
      { One of the classes of objects between which data communications take
      place. }
  end Communicant;

  object DataBus is
    components: ;
    operations: TransmitData(Communicant, Communicant, DataBus): boolean
    description:
  end DataBus;

  {**** Control System Computation Requirements (F.C.C.1), Pg. 223 ****}

  let cm1, cm2: Communicant;
      db: DataBus
  axiom (forall cm1, cm2, db :
    if TransmitData(cm1, cm2, db)
    then (db.Type = Electrical or db.Type = Optical) and
         (db.Speed > MinimumDataCommSpeed)

end FlightControlSystem;
```

C Analysis of the Translation of ASCT into WSRSL

This section of the report contains a page-by-page analysis of the translation of ASCT into the WSRSL requirements specification, the latter hereafter referred to as "the spec."

Page i: Preface. Introductory comments such as this can appear as comments in the top-most module of the spec.

Page iii: Table of Contents. In general, tables of contents in textual documents are replaced by the top-level roadmap available in the electronic browser.

Page 1: 1.0 Introduction. More introductory comments.

Pages 3-5: 2.0 Requirements Generation. Some of the commentary appearing in this section is appropriate to put in the top-level module commentary. Much of the commentary pertains to the specific techniques used in the development of ASCT, which techniques are not applicable to the WSRSL-developed document.

In general, "meta-commentary" about the development language itself, rather than about the contents of a particular specification are not included in the specification itself but rather in accompanying documents describing the WSRSL language and methodology.

Page 7: 3.0 Excelerator/RTS Overview. Not applicable to the WSRSL spec.

Page 9: 4.0 Advanced Flight Control System Requirements. Remarks made about pages iii and 3-5 apply here.

Page 10: Figure 1 Organization of Control Functions ... . This is the top level of DFD, which is browseable in DFD tool. In general, the WSRSL Browser allows the DFDs to be used as a functional roadmap for the requirements specification, which is a significant advantage over the paper-based ASCT document.

Page 11: Figure 2 Reports Generated ... . The precise names of reports listed in this figure are based on those produced by Excelerator, and are therefore not exactly applicable to the WSRSL specification. In general, the hypertext capabilities of the WSRSL Browser allow any number of "reports" to be viewed, by establishing the appropriate document links. This is a significant advantage of an electronic browser over a paper document.

Pages 12-13: Fly Mission DFD and descriptions. These and all other detailed DFDs are represented in electronic, browseable form in the WSRSL spec. The textual process and dataflow descriptions appear in the description fields of the formal object and operation definitions. As noted above, the browseable configuration of the WSRSL spec permits the user to move freely from graphical to textual descriptions, enhancing the connectivity of the document.

It is worth noting that there is a syntactic error in the data flow description on Page 13, with respect to the DFD appearing on Page 12. Specifically, the name dataflow name "Mission" appears in the DFD, whereas the corresponding name in the textual description is "Mission Definition". Similar consistency errors occur elsewhere in ASCT. Since the ASCT DFDs were produced by Excelerator, it is not clear just how these errors arose, since it is our understanding that Excelerator should generate data dictionaries directly from DFDs, thereby precluding such errors. At any rate, such errors should be impossible in any system where graphical depictions are derived directly from formal definitions. This is the ultimate goal for the WSRSL browser, which we plan to implement next year.

Page 14: Figure Mission.Analysis.1 Mission Segments. In the formal WSRSL spec, what appears in this figure is a graphical view of the Mission object. As described in our proposal for the coming year, views are mapped formally to specification objects. Such mappings are advantages in that they allow graphical depictions of objects to be generated formally from object definitions. We hope to implement a prototype of the language support tool that will allow formal specification of graphical components in a requirements specification document.

Page 15: Table 1. Analysis of Mission Segments. This table is very interesting in terms of what it indicates about the completeness of the ASCT document. As noted in Section 4 of the report, the ASCT document, organized around a functional DFD description, lacks a formal description of most of the major objects that are mentioned in the requirements statement. In particular, the Mission Segments description presented in Table 1 is in fact a form of object description. This is reflected in the WSRSL spec by the appearance of object Mission. What is noteworthy about Table 1 is that it is one of very few ***explicit*** object definitions appearing in the ASCT document. The component details of almost all other object definitions in the WSRSL spec were gleaned from various textual "clues" found in ASCT requirements and supporting tabular information.

Table 1 on page 15 contains a considerable amount of information, in a somewhat ad hoc format. WSRSL was used to organize the information slightly more formally, by defining object fields "control_action", "driver", and "control system requirement". However, these fields are semantically comments, the text being copied directly from Page 15. A more formal and precise decomposition of these requirements would involve further object/operation decomposition and formal requirements specification. For example, the "control_system_requirement" for the Take-Off phase of the mission is a rather long list of entities, the relationship of which to TakeOff is not particularly clear. It appears likely that these requirements would be best specified formally as preconditions to a TakeOff operation. Similar formal analysis of the rest of the "control actions", "drivers", and "control system requirements" is in order.

Page 16: Table 2. Assignment of Control Requirements to Functions. This is a somewhat ad hoc assemblage of information that is factored into the WSRSL specification in a less ad hoc manner. What is conveyed in Table 2 is the connection between requirements and specification operations. In a WSRSL spec, this connection is made formally according to the module context in which a requirements statement appears. Specifically, when a requirement is stated as a pre or postcondition to an operation, then it is directly connected to that operation. When a requirement is stated as a module invariant, then it is associated with all of the operations of the module.

Page 17: Cntrl.Mission.Flight Req.List. This is a reasonably useful piece of information that can be represented in the WSRSL spec in a number of ways. In the current WSRSL version of ASCT, it is represented in two manners. First, an operation field "CMF" is defined that is used to record the the names of requirements within the operations to which they apply. This style of requirement reference could be carried out within the rest of the spec (but it is not done so in the current draft).

The other representation of requirements relations is as thread of hyperlinks that connect each group of requirements. These links are used within the browser to navigate from entities to associated requirements and vice versa.

Another alternative to this organization of the top-level requirements list would organize the modular structure of the WSRSL spec around requirements rather than functionally. This organization would be inferior to the current functional organization, however, since it would spread object and operation definitions across modules in a non-functional way.

Page 18: General Control Requirements (C.M.F.1). This is the first of the actual requirements statements appearing in ASCT. As described in Section 4, these requirements are factored throughout the WSRSL version of the spec as object definitions, operation definitions, pre and post conditions, and module axioms. Here in C.M.F.1 the requirements make reference to the handling quality criteria for the aircraft. In translating to a more formal WSRSL statement of C.M.F.1, the ambiguous terms used to define handling quality must be formalized. For example, in the statement "... without requiring exceptional pilot skill or strength", the "exceptional" must be quantified, and "skill" and "strength" must be formally identified as attributes of a pilot. Hence, we define a Pilot object with attributes Strength and Skill, provide some quantitative (e.g., numeric) value for these attributes, and then state requirements about these attributes formally.

It should be noted that detailed object descriptions of crew are missing from the original ASCT document. The addition of such object definitions in the WSRSL document is essential to the formalization of requirements.

The representation of C.M.F.1 in WSRSL in representative of one strategy for stating temporal transitions. That is, the WSRSL statement of C.M.F.1 is representative of how similar requirements throughout the WSRSL spec can be stated when the requirements express what happens when a state of the aircraft changes. Specifically, the technique used is based on axioms that use existential quantification

```
time: type ...
Mission: type ...
Aircraft: type ...

t1, t2: variable time
m: variable Mission
a: variable Aircraft

p: function[sometype,... -> boolean]

A1: axiom
  if (exists t1, m : m.Time = t1 and a.State.HandlingQuality
```

Page 85: Flutter Prevention Requirements (C.M.F.26). Regarding the statement "The airplane shall comply with ... FAR 25.629", in a fully formalized document, references to external requirements such as FAR must ultimately be formally represented. While this will require a potentially very large effort, it is necessary in order to ensure that requirements are validatable.

Throughout ASCT, there are requirements of the form "It shall be shown by analysis of tests ..." Such statements can be considered a form of ***meta-requirements*** in that they specify how other requirements are to be met.

Page 87: Control Mission Flight DFD.

Page 88: Control Mission Flight DFD Component Descriptions. Since this is the top-level DFD, it contains reference to several key top-level objects. As has been noted earlier, one of the major deficiencies of operation-oriented dataflow analysis is the tendency to under define system objects. This is most obvious at the highest level DFDs. Hence, in the WSRSL spec, most of the objects defined on page 88 of ASCT appear in other object-oriented modules (e.g., ActualFlightPath appears in module FlyMission). This reorganization of system objects into object-oriented modules is the single largest structural difference between ASCT and the WSRSL translation.

Page 105: ControlAerodynamicBraking DFD. Here and elsewhere in the document the use of transforms labeled "Display..." are somewhat ad hoc. In general, if such transforms appear explicitly anywhere in the specification, they should appear consistently throughout the DFDs for all aspects of Mission control that are displayed to the end users (i.e., the crew). There are certainly many aspects of mission control that are displayed to the crew that do not appear in any ASCT DFD. The point is not to belittle the original document, but to point out the necessity for completeness and consistency with regards to user interface aspects of the requirements.

An alternate methodology for specifying user interface components and requirements is that discussed in the proposal for next year's project. Using this methodology, user interface aspects are factored completely out of the abstract functional requirements specification, but still defined formally to permit verification and validation.

Page 127: Generate Flight Path Command DFD. This is the first DFD in which the ***control transform*** notation is used. It is not clear from the context of use, nor from the explanation of control transforms in Appendix B, how the distinction between process versus control transform is being exploited in the requirements at this point. In WSRSL, both control and process transforms are represented as operations.

It is arguable that the difference between a specification and an implementation is the internal operational details of operations. While the distinction between a process transform and control transform can be useful for modeling an executable prototype or implementation, the interpretation of intra-operation behavior may well be considered outside the scope of a functional requirements specification. This point is debated by researchers and practitioners in software engineering. If the goal for a requirements specification is to make it more formal and verifiable, we think it best to draw the line at the internal behavior of operations. That is, this operational behavior should not be specified via the process description techniques described in Appendix B of ASCT.

The previous paragraph is not intended to indicate that we do not believe in the concept of a wide spectrum language that can be used for both specification and design, just that the line should be drawn ***above*** operational details in a requirements specification.

Operational models, such as those presented in Appendix B, can certainly be useful as a requirements specification is refined to a system design and implementation. In the specific context and scope of ASCT, however, it is not clear that the level of abstraction represented ***within*** process versus control transforms provides any tangible support for specifying requirements. This is born out by the lack of detailed control information that appears in the ASCT within the requirements themselves. That is, the detailed discussion in Appendix B of how to implement control transforms does not appear to be used anywhere in the body of the document.

Page 129: FlightControlSystemPitchFunctions. This first DFD in which the use of ***Architectural Elements*** appears. It is our understanding based on the description given in Appendix C, that Architectural Entities represent essentially objects in WSRSL.

One of the long-standing deficiencies of requirements expressed in the DFD-based structured analysis models was the second-class treatment of object descriptions. As noted in Section 4 of the report, this deficiency was addressed in requirements specification languages such as RSL, in which objects and operations are given equal treatment. Hence, a requirements specification should be viewed from ***both*** an operation-oriented perspective ***and*** an object-oriented perspective.

The other important requirements concept that appears to be addressed by the use of AE's is the specification of the ***agent*** of an operation. As described in the main body of the report, an agent is formally nothing more than an object.

The formal semantics of an agent can be represented as a higher-order agent-evaluation function that takes as inputs (a) the agent object (b) the operation that the agent is to execute, and (c) the inputs to the operation. In this sense, and agent can be considered an meta-operation – an operation that performs another operation.

In order to effect a fully formal representation for architectural entities in WSRSL, the following idiom is used,

```
object SomeAE is
  components: ... ;
  operations: ..., PerformSomeAEFunctions;
  description:
     The object definition that corresponds to an ASCt architectural entity ;
end SomeAE

operation PerformSomeAEFunctions is
  components: ... ;
  inputs: ... ;
  outputs: ... ;
  agent: SomeAE;
  description: ;
     The operation performed by the SomeAE.
end PerformSomeAEFunctions
```

While this idiom is obvious, it is important since it defines formally how the values of an AE are produced. The ASCT use of AE's suggests an object that produces values, which cannot be formally represented, since only operations (based on EHDM functions) can produce values.

The existence of AE's relates to a subtle conceptual point that arises frequently in formalizing requirements specifications. Viz., what is the difference between a behavior producing ***object*** and the ***operation*** that denotes the behavior it produces. Consider, for example, a Pilot. What is the difference, conceptually, between the Pilot as an object and a Pilot as a function? Clearly, a pilot can be viewed as an object, with components, such as specific anatomical parts, that may be relevant to a requirements specification. Equally clearly, a Pilot can be viewed as a operation, since the behavior of the pilot produces and uses values that are the outputs to and inputs from other operations in the complete operational aircraft system.

The answer in the WSRSL version of ASCT is to view entities such as pilots as both objects and operations, using the naming idiom illustrated above. This provides the fully formal representation of an Architectural Element as used in ASCT.

Page 133 Flight Control Sys Pitch Functions DFD. This DFD is the first occurrence of operations of the for "ProvideXXXInterface", where "XXX" is some crew member. Operations of this nature suggest to a limited extent what the end user (i.e., human) system interface should be. However, there is much more to the human-system interface than feedback operations such as these. In fact, we argue that consideration of the human interface is a critical aspect. As noted earlier, this topic is discussed in the proposal for the follow-on project.

Page 134: Process descriptions for Flight Control Sys Pitch Functions. The description of DisplayPitchEnvelopeProtectStatus is somewhat curious ...

Page 157: Process descriptions for Flight Control Sys Roll Functions. See comments above on Page 134.

Page 199: F.C.S.1 This is another example of a requirement stated in prose that leads to the definition of further objects and/or operations than were defined explicitly in ASCT. Specifically, the objects Communicant and DataBus were added.

Report Documentation Page

1. Report No. NASA CR - 187522	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle: Structured Representation for Requirements and Specifications		5. Report Date: July 18, 1991
		6. Performing Organization Code
7. Author(s): Gerald C. Cohen, Gene Fisher, Deborah Frincke, Dave Wolber		8. Performing Organization Report No.
		10. Work Unit No. 505-64-10-07
9. Performing Organization Name and Address: Boeing Military Airplanes, P.O. Box 3707, M/S TJ-24, Seattle, Wash 98124-2207		11. Contract or Grant No. NAS1-18586
12. Sponsoring Agency Name and Address: NASA Langley Research Center, Hampton, VA 23665-5225		13. Type of Report and Period Covered: Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Sally Johnson

Task 2 Report

16. Abstract

This document vas generated in support of NASA contract NAS1-18586, Design and Validation of Digital Flight Control Systems suitable for Fly-By-Wire Applications, Task Assignment 2. Task 2 is associated with a formal representation of requirements and specifications. In particular, this document contains results associated with the development of a Wide-Spectrum Requirements Specification Language (WSRSL) that can be used to express system requirements and specifications in both stylized and formal forms. Included with this development are prototype tools to support the specification language. In addition a preliminary requirements specification methodology based on the WSRSL has been developed. Lastly, the methodology has been applied to an Advanced Subsonic Civil Transport Flight Control System.

17. Key Words (Suggested by Author(s)): Formal Requirements and Specification; Wide-Spectrum Requirements Specification Language (WSRSL); WSRSL Parser; Dataflow Tool; WSRSL Browser; Hypertext	18. Distribution Statement: Unclassified - Unlimited. Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	88	